A LOGICAL DESIGN AUTOMATION UTILITY

Chin-Chi Kao, et al

Washington University

Prepared for:

Advanced Research Projects Agency
Department of Defense
Public Health Services

July 1970

Unclassified
Security Classification

AD 771748

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Computer Systems Laboratory
Washington University
St. Louis, Missouri

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE
A Logical Design Automation Utility

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Interim

5. AUTHOR(S) *(First name, middle initial, last name)*
Chin-chi Kao and Ying Huang Chuang

6. REPORT DATE
July, 1970

7a. TOTAL NO. OF PAGES
72 93

7b. NO. OF REFS
22

8a. CONTRACT OR GRANT NO.
(1) DOD(ARPA) Contract SD-302
(2) NIH(DRFR) Grant No. 00396

b. PROJECT NO.
(1) ARPA Project Code No. 5880

c. Order No. 655

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
Technical Report No. 19

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT
Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
ARPA – Information Processing Techniques,
Washington D.C. NIH., Div. of Research

13. ABSTRACT

Computer oriented algorithms for several laborious computations frequently encountered in switching theory and logic design are presented. They are algorithms for the computation of designation numbers, functional composition, detection of relations between Boolean functions, symbolic expansion of Boolean expressions, and approximate minimization of Boolean functions. These algorithms are useful in the construction of man/machine interactive systems for logic design automation. Machine independence and modularity are emphasized in the development of these algorithms. They have been programmed on the LINC computer.

DD FORM 1 NOV 68 1473 REPLACES DD FORM 1473, 1 JAN 64, WHICH IS OBSOLETE FOR ARMY USE.

ia

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Logic Design Automation						
Switching Theory						
Computer Oriented Algoritlms						
Designation Numbers						
Reverse Polish Strings						
Function Equivalence						
Logic Minimization						
Symbolic Expansion						

A LOGICAL DESIGN AUTOMATION UTILITY

Chin-chi Kao and Ying Huang Chuang

TECHNICAL REPORT NO. 19

July, 1970

Computer Systems Laboratory

Washington University

St. Louis, Missouri

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract SD-302 and by the Division of Research Facilities and Resources of the National Institute of Health under Grant RR-00396.

ABSTRACT

Computer oriented algorithms for several laborious computations frequently encountered in switching theory and logic design are presented. They are algorithms for the computation of designation numbers, functional composition, detection of relations between Boolean functions, symbolic expansion of Boolean expressions, and approximate minimization of Boolean functions. These algorithms are useful in the construction of man/machine interactive systems for logic design automation. Machine independence and modularity are emphasized in the development of these algorithms. They have been programmed on the LINC computer.

TABLE OF CONTENTS

TABLE OF CONTENTS

(continued)

LIST OF TABLES

LIST OF FIGURES

A LOGICAL DESIGN AUTOMATION UTILITY

1. INTRODUCTION

Due to their great computational speed and simulation ability, digital computers are being used in designing and analyzing new computers, and this application is becoming wider. Breuer[1] has divided the problems in design automation of digital computers into three main areas: Preconstruction analysis, hardware design and implementation, and software generation. Preconstruction analysis deals primarily with the use of general purpose system simulators in trying to determine an optimal functional configuration for a new computer system[2]. Main problems under hardware design include automatic logic generation, Boolean logic minimization, and logic simulations. Problems in physical implementation include printed circuit layout, the assignment of circuit cards to backboards, the placement of circuit cards on backboards, and the interconnection of terminals, etc. Typical problems in automatic software generation include the use of a meta-compiler as an aid in generating a new compiler, and automatic generation of test routines for detection and diagnosis of component failures[3,4].

In general, the usage of digital computers in hardware design and implementation has been primarily concerned with the generation and maintenance of files for recording logic network structures, placement of logic cards, development of wiring lists, and other manufacturing oriented data[5]. Such programs do not alter the engineer's logic design, but merely check its validity under certain manufacturing constraints and develop appropriate fabrication information. Although great interest has been expressed in developing computer algorithms for automating tasks of logic design, only the feasibility has been demonstrated[6]. With the increasing complexity of digital systems and resulting need for greater control and accuracy in the design phase from architecture to implementation, it is crucial that efficient algorithms for automated logic design be developed and implemented.

The purpose of this research is to develop some computer-oriented algorithms for the manipulation of switching functions in logic design. Although some computer algorithms have been developed to manipulate switching functions[6,7], most of these are for special purposes and dependent on the particular systems and languages used by the developers. Hence the algorithms are not easily applicable without having the same types of facilities. In this development, however, particular emphasis has been put on modularity and machine independence. It is certain that the users can easily program these algorithms on their own computers.

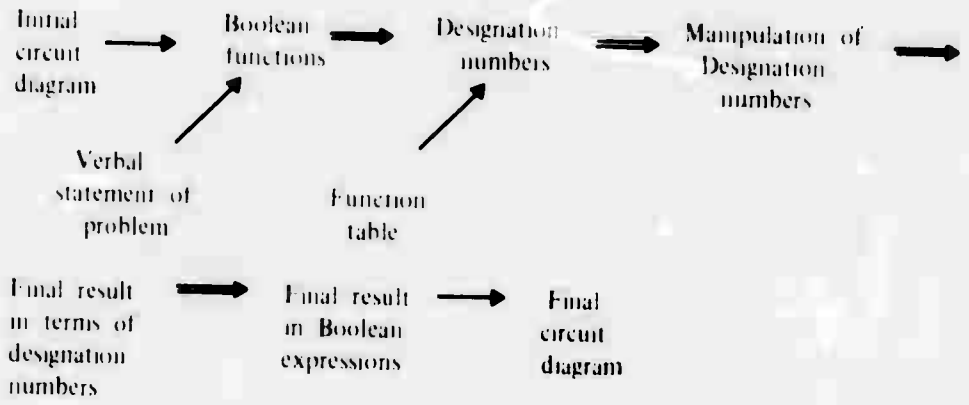
2. DESIGNATION NUMBERS

2.1 INTRODUCTION

Boolean algebra can be applied to describe relationship between input and output variables of a *switching function*. To write a *Boolean expression* describing the input-output relation of a switching circuit, however, two problems appear. Firstly, there are in general many Boolean expressions which can describe the same switching function. Secondly, the algebraic manipulation of Boolean expression not only requires considerable skill and imagination together with a thorough knowledge of Boolean algebra, but is also difficult to be mechanized.

The solution to the first problem is a representation which is unique for a given function. While the solution to the second problem is a concise and convenient representation feasible for mechanical computation. A representation which can solve these two problems simultaneously is the designation number[8].

The role played by designation number in logic design process can be expressed in the following:



The Roles of the Designation number in Logic Design

Figure 1

In this figure, the double-lined arrows refer to problems that involve only Boolean algebra and the computation methods; the single-lined arrows refer to procedures unique to the digital computer circuit problem.

2.2 TABLE OF COMBINATIONS AND DESIGNATION NUMBERS

Input-output relation of a Boolean function can be expressed by using a table of combinations which is a table listing values of the function for all possible combinations of values of input variables. For instance, the

switching function

$$F(ABC) = AB+C'$$

has the table of combinations given in Table 1.

	A	B	C	AB+C
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Table 1
The Table of Combinations of the Function F(ABC) = AB+C'

In the table, input combinations are listed according to the value of their BCD (binary coded decimal). When this natural order is understood, the last column (i.e., the column of function values) only is sufficient to describe the function. This column of binary digits taken as a binary number is called the ***designation number*** of the Boolean function.

Similarly the column of binary digits under an input variable in the table is called the designation number of the input variable. Thus the designation number of the input variable C is 01010101 and that of the function F is 10101011. For a function of n variables, the designation numbers of the function and input variables have 2^n digits because there are 2^n combinations of the values of input variables.

If one changes the order of input variables, the designation numbers of input variables have to be changed, and the designation number of the function changes consequently. Therefore, when we talk about the designation number of a Boolean function, the sequence of input variables must be clearly specified. With the order of variables specified, the designation number is a unique and compact representation of a Boolean function.

2.3 COMPUTING THE DESIGNATION NUMBER OF A BOOLEAN FUNCTION

The designation number of a function can be computed by substituting designation numbers of input variables into the Boolean expression describing the function and then perform logical operations bit by bit. The order in

which logical operations are performed is specified by the explicit and implicit hierarchy in the syntax of the expression. For manual computation, this order is not difficult to see. For machine computation, however, it is much more efficient if the order of logical operation is the order of appearance of operator symbols in the expression. Therefore, the first step in this computation is to transform the given Boolean expression – generally in infix notation – into a reverse Polish form.

2.3.1 Boolean Expression in Infix Notations

A Boolean expression in infix notation is defined recursively as follows:

1. Any alphabetic letter is a Boolean expression.
2. If A and B are Boolean expressions, then so are (A+B) and (AB).
3. If A is a Boolean expression, then so is A'.

Only symbol strings satisfying the above rules are Boolean expressions in infix notation and no others. A+B, AB, and A' mean A "OR" B, A "AND" B, and A "NOT" respectively. Parenthesis pairs can be omitted if their omission does not give rise to ambiguity. Thus the symbolic convention chosen here is the one most commonly used.

Consider the Boolean expression in infix notation

$$F(ABCDEHKLMN) = (A + CD + C(D+E)'(M+N) + H + K)'L$$

To compute the designation number of this function, one can perform the logical operations in the following order:*

1. C D
2. A + CD
3. D + E
4. (D + E)'
5. C · (D + E)'
6. M + N
7. C(D + E)' · (M + N)
8. (A + CD) + C(D + E)'(M + N)
9. (A + CD + C(D + E)'(M + N)) + H
10. X = (A + CD + C(D + E)' (M + N) + H) + K
11. X'
12. X'L

*This order is the one implied in the reverse Polish string obtained using the algorithm of the next subsection.

Unless the expression is in sum-of-products form or product-of-sums form (namely, normal forms), the procedure of finding the order of logical operation is difficult. Since the expression given is generally not in a normal form, either transformation to a normal form or to a reverse Polish string is required. A normal form transformation, namely expansion of Boolean expressions, will be discussed in Chapter 5.

The algorithm to transform a Boolean expression into a reverse Polish form is a slight modification of the well known algorithm which transforms an ordinary arithmetic expression into its reverse Polish form[9]. This algorithm is discussed in the following subsection.

2.3.2 Converting Boolean Expression into Early Reverse Polish Form

A Boolean expression in early reverse Polish form (ERP) is defined recursively as follows;

1. Any alphabetic letter is an ERP form.
2. If A and B are ERP forms, then so are AB+, AB·, and A′.
3. There are no others.

The use of the reverse Polish notation to represent a Boolean expression has the following advantages:

1. The operators appear in the order in which the computation is carried out.
2. It eliminates the necessity of parentheses.

The early reverse Polish notation rather than late reverse Polish is adopted, because the mechanization of transforming to the ERP form requires smaller push-down store.

The essential rules of converting Boolean expression into ERP notation are as follows:

1. If s_j is a variable, it is transcribed directly to the output.

2. If s_j is a left parenthesis following a variable, a "NOT" operator, or a right parenthesis, first a logical "AND" sign "·" and then a left parenthesis are put to the push-down list N.

3. If s_j is an operator, the top entry (called E) of the list N is examined. If E is an operator not weaker than s_j in the operator precedence given in the list below, then E is transcribed to the output. The process repeats for each top entry of the push-down list N until it is empty, or the top entry is a left parenthesis or an operator weaker than s_j. When this happens s_j is transcribed to the push-down list N.

The List of Operator Precedence

1. ′ (NOT)
2. · (AND)
3. + (OR)
4. ⊕ (Exclusive-OR)

4. If s_i is a right parenthesis, successive top entries are transcribed from the list N to the output until a left parenthesis is reached, which is then deleted.

5. If a variable or a complemented variable is following a variable, a "NOT" operator, or a right parenthesis, then a "." is put to the output.

6. After the last symbol of the Boolean expression has been dealt with, the remaining entries in the list N are transcribed to the output.

In general, the symbols of the Boolean expression are examined one by one from left to right. Variable symbols are transcribed as soon as they are encountered. Operator symbols are held in a push-down store N until conditions for their transcription are satisfied. The process is more clearly described in Appendix 8.1.

2.4 PROGRAM IMPLEMENTATION

2.4.1 The Program to Obtain an Early Reverse Polish String

The algorithm for changing an Boolean expression to an early reverse Polish string is shown in Appendix 8.1.

An operator code and its hierarchy number are stored alongside in the push-down list N. The hierarchy number for the unary operator "NOT" is 14, while those for binary operators are "AND" 12, "OR" 10, and "Exclusive-OR" 8. The hierarchy numbers for the right and the left parentheses are 04 and 06 respectively.

To make sure that the input Boolean expression is well-formed, the following rule is used to check the reverse Polish string obtained. This is because the input Boolean expression is well-formed if and only if the reverse Polish string obtained is also well-formed. Each symbol in the string is given a weight. The weights of binary operators "AND," "OR," and "Exclusive-OR" are 1; that of unary operator "NOT" is 0; and that of any specified variable is −1. Each unspecified symbol is arbitrary given a weight of 100. Then the reverse Polish string is well-formed if and only if the sum of the weight of all symbols in every **proper tail** of the string is nonnegative, and the sum of the weight of all symbols in the whole string is −1. A tail of a string w = xy is y, and y is proper if x is not a null string.

2.4.2 The program to Obtain a Designation Number from a Reverse Polish String

In performing "AND," "OR," "NOT," and "Exclusive-OR" operations, each variable of the string is represented by a vector (designation number). The length of the vectors depends on the number of variables of the function. Each vector of a k-variable function have 2^k binary bits. Though each word of the LINC has 12 bits, only the upper 8 bits are used. For byte oriented machines, there will be no memory waste. We could construct and store these vectors in their full length, and use them in the computation of designation numbers. However, since these vectors are generally very long, it would be very uneconomical both in time and in memory space. Therefore, we divide each vector into 8-bit (i.e., 1-word) sections, and calculate successive sections of these vectors in each iteration of the process. This is possible because bit patterns of the vectors change in a regular way. The first, second, and third variable vectors repeat with the pattern of 01010101, 00110011 and 00001111

respectively. Vectors from the fourth variable and on are combinations of $\bar{0}$ (00000000) and $\bar{1}$ (11111111) in a simple and regular way.

For a Boolean function of n variables, the computation of designation number is divided into $m = 2^{n-3}$ iterations. In the first iteration, the reverse Polish string is scanned from the left end to the right end, and each time an operator is met an operation is executed using the first words of variable vectors only. Similarly, in the second iteration, operations are executed using the second words of variable vectors only. The whole computation ends at the end of the m-th iteration. In each iteration, the string is scanned from the beginning to the end, and the section of each variable vector calculated as follows.

A register is provided for iteration count and for indicating whether the 8-bit sections of the fourth to fifteenth variable vectors in the current iteration is $\bar{0}$ or $\bar{1}$. Bit 0 through bit 11 indicate the bit patterns of the fourth through the fifteenth variable vectors respectively. Thus a 0 (or 1) in the i-th bit indicates that the i+3rd variable vector is $\bar{0}$ (or $\bar{1}$) in the current iteration. The content of the register is set to 000000000000 in the first iteration, and indexed by 1 after each iteration. For example, in the tenth iteration the content of the register is 000000001001, which indicates that the vectors of the fourth and the seventh variables are $\bar{1}$'s, while those of other variables except the first, second, and the third are $\bar{0}$'s.

This approach has an additional advantage of not requiring calculation of variable vectors each time the number of variable is changed.

2.5 COMPOSITE FUNCTIONS

Most switching circuits are composed of several stages. The inputs to higher stages are not all pure variables (i.e., primary input variables); some of them are the outputs of previous stages. The block diagram of Figure 2 shows a circuit composed of two stages and three component circuits f_1, f_2, and f_3. The inputs of circuits f_1 and f_2 are pure variables A, B, and C, and their outputs are labeled f_1 and f_2 respectively.

Therefore, $f_1 = f_1(A,B,C)$ (2.5.1)

and $f_2 = f_2(A,B,C)$ (2.5.2)

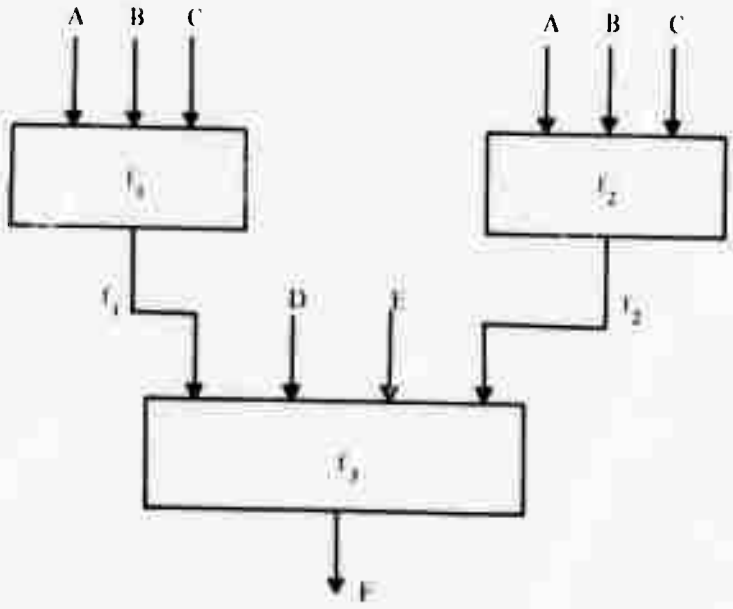
The circuit f_3 has inputs f_1, f_2, D, E, and output labeled F.

$$F = F(f_1, f_2, D, E) \quad (2.5.3)$$

is the composite function, because f_1, f_2 are not pure variables; F is a function of pure variables A, B, C, D, and E because f_1, f_2 are functions of A, B, and C. To obtain the designation number of F(A,B,C,D,E) we substitute equations (2.5.1) and (2.5.2) into equation (2.5.3) first, and then calculate the designation number of the resulting expression. A program is developed for this substitution.

2.6 REMARK

By translating a Boolean expression into an early reverse Polish string, the computation of the designation number for any Boolean expression containing "AND," "OR," "NOT," and "Exclusive-OR" operators becomes



A Composite Circuit Composed of Two Stages
and Three Components

Figure 2

simple. However, because the procedure to compute the designation number of a function given in sum-of-products form is also pretty straight forward, we can expand the Boolean expression into a sum-of-products form instead of translating into a reverse Polish form. For Boolean expressions containing the "Exclusive-OR" operator, however, the normal form expansion becomes very complicated. Therefore, the algorithm based on reverse Polish translation is a much reasonable approach.

3. FUNCTIONAL RELATIONS

3.1 INTRODUCTION

Designation numbers can be used effectively to find relations between functions. In this chapter, we shall discuss how designation numbers can be used to achieve this.

Many different expressions can represent the same function. But given a variable sequence, all different expressions of the same function have a unique designation number. *Equality* of Boolean expressions can, therefore, be detected by comparing their designation numbers.

Two different functions have different designation numbers based on the same variable sequence. Very often, however, different functions are considered to be *equivalent*, if their designation numbers can be made equal by fixing the variable sequence of one function while changing the variable sequence of the other function in all possible ways. For example, the functions (with the variable sequence shown inside the parenthesis pair)

$$F(ABCD) = A + B$$

$$G(ABCD) = C + D$$

are different. Designation numbers of F and G are 00001111 11111111 and 01110111 01110111 respectively. These two functions are equivalent, however, because the designation number of F(ABCD) is identical to the designation number of G(CDAB). Detection of functional equivalence is important in logic design, because equivalent functions can be realized by identical networks.*

Other functional relations such as the implication relation can also be detected by comparing designation numbers.

3.2 EQUALITY AND IMPLICATION

Equal functions have identical designation numbers based on the same variable sequence. Therefore, we can easily detect equality of two functions by comparing to see if their designation numbers match.

The implication relation between two functions is also detectable by comparison of their designation numbers. F implies G if and only if in all bit positions in which F has a 1, G also has a 1. The detection of implication relation is useful in residue test[10].

3.3 FUNCTIONAL EQUIVALENCE

Although detection of functional equivalence is very important in logic design, there is still no efficient detection algorithm available at present. Before a more efficient algorithm is available, the best we can do to detect functional equivalence is by comparing designation numbers with the variable sequence of one of the

*Functional equivalence obtainable by complementing variables is not considered here.

functions varying in all possible ways. This exhaustive approach becomes feasible only with the aid of high speed digital computers.

The essential part of functional equivalence detection is, therefore, the algorithm of computing all possible permutations of a variable sequence. The method developed by K. Harada[11] is adopted here. This method generates all Hamiltonian circuits of a non-oriented complete graph. The variables of a Boolean function are considered as the vertices of the graph, and each Hamiltonian circuit corresponds to a variable sequence. All permutations of variables can be obtained by rotating variable sequences corresponding to all Hamiltonian circuits. For a graph with n vertices, there are $\frac{(n-1)!}{2}$ Hamiltonian circuits. The $\frac{(n-1)!}{2}$ variable sequences corresponding to these Hamiltonian circuits are then rotated n times clockwise and n times counterclockwise to obtain $\frac{(n-1)!}{2}$ (n+n) = n! permutations without duplication or omission.

Suppose functions F and G have the same number of variables and their equivalence is to be detected. The designation number of F (with a fixed variable sequence) is computed with that of G (with the same variable sequence as that of F initially). If their designation numbers are different, a new variable sequence and the corresponding designation number are computed for G. The new designation number of G is then compared with the designation number of F. This procedure continues until the designation number of F is equal to a designation number of G, in which case F and G are equivalent; or all designation numbers of G corresponding to all possible permutations of its variable sequence have been compared, in which case F and G are not equivalent.

3.4 COMPARING BOOLEAN FUNCTIONS WITH DON'T CARES

A Boolean function may have don't care conditions. These don't care conditions can be specified in the form of a Boolean function called don't care function having the same variables as the given Boolean function. A function with don't care conditions, therefore, consists of two component functions: a *care function* which specifies input combinations for which the function has value 1, and a *don't care function* which specifies don't care input combinations. Denote the care function and the don't care function of a function F by F_c and F_d respectively. To compare F and G for equality, implication or equivalence, we compute designation numbers of F_c, F_d, G_c, and G_d separately. Designation numbers of F_d and G_d are added together (component-wise ORing) to form a mask. We then compare designation numbers of F_c and G_c ignoring bit positions in which the mask has 1.

3.5 EXAMPLE

Consider the two functions F and G with $F_c(ABCD) = AC + BD + CD$, $G_c(ABCD) = AB + AC + B'CD + BC'D$, $F_d(ABCD) = A'BCD' + AB'C'D$, and $G_d(ABCD) = A'BC$. Denote the designation number of a function by adding a # sign before the function symbol. For instance, the designation number of $F_c(ABCD)$ is denoted by $\#F_c(ABCD)$. Then,

$$\#F_c(ABCD) = 00010101\ 00110111,$$

$$\#F_d(ABCD) = 00000010\ 01000000,$$

$$\#G_c(ABCD) = 00010100\ 00111111,$$
$$\#G_d(ABCD) = 00000011\ 00000000.$$

The mask is 00000011 01000000.

By comparing $\#F_c(ABCD)$ and $\#G_c(ABCD)$ with mask bits (i.e., bit positions in which the mask has a 1) ignored, we see that F and G are not equal but F implies G.

When the variable sequence of G is changed to ADBC, the designation numbers of G_c and G_d are

$$\#G_c(ADBC) = 00000110\ 01110111,$$
$$\#G_d(ADBC) = 00010001\ 00000000.$$

The mask, which is the bit-wise ORing of $\#F_d(ABCD)$ and $\#G_d(ADBC)$, is 00010011 01000000. Under this new mask, $\#F_c(ABCD) = \#G_c(ADBC)$. Therefore, F and G are equivalent.

4. BOOLEAN EXPRESSION MINIMIZATION

4.1 INTRODUCTION

The design of a combinational logical network generally involves two steps. In the first step, the desired relation between the input and output signals is stipulated, from which a Boolean expression which represents the network function is derived. There are, in general, numerous different Boolean expressions which are equivalent in that they all represent the same network function. Therefore, in the second step, calculation may be performed on the expression obtained in the first step to obtain the "minimum" equivalent expression. This second step is termed the minimization or the simplification process.

An equivalent expression is a minimal expression if one of the following three minimality criteria is satisfied.

1. The minimal expression is the expression with fewest literals.

2. The minimal expression is the expression with minimal sum of literals and terms.

3. The minimal expression is the expression with the fewest terms, provided another expression does not have the same number of terms and fewer literals.

Quine[12] showed that under any such criterion, the minimal expression is a disjunction of certain products called *prime implicants*. The minimization process, therefore, involves first obtaining the set of prime implicants and then choosing the best irredundant set of prime implicants based on the minimality criterion used.

The process of obtaining the absolute minimum expression is a complex and time consuming computation process, mainly because for some expressions the number of prime implicants is very big[16]. It is quite often the case that the hardware cost saved by finding a minimum circuit is far less than the cost of finding that circuit, and a nearly minimum, not the absolute minimum circuit, which is rapidly found is the most economical. Thus our primary concern is to find an algorithm for obtaining an approximate minimum form with emphasis on the speed of computation and memory requirement.

The techniques of minimizing Boolean expression can be classified as map methods[14], tabular methods[12, 13] and cubical complex methods[15, 16]. Map methods are satisfactory for functions of no more than six variables. The tabular method and the cubical complex method are more suitable to machine computation. Moreover, the usefulness of these two methods are not limited by the number of variables. The approximate minimum method which is the main subject of this chapter stems from the cubical complex method. Therefore, this chapter will begin with a brief discussion of cubical complexes.

4.2 CUBICAL COMPLEXES

A geometric representation of a Boolean function is obtained by mapping a Boolean function of n-variables onto the n-dimensional unit cube. We set up a coordinate system on the n-cube with coordinates $(e_1, e_2, ----, e_n)$

where $e_i = 0, 1$. We then make the correspondence between the fundamental product $x_1^{e_1} x_2^{e_2} \cdots x_n^{e_n}$ and a vertex $(e_1, e_2, \cdots, e_n)$, where $x_i^{e_i} = x_i$ if $e_i = 1$, and $x_i^{e_i} = x_i'$ if $e_i = 0$.

Let Z_2 represent the space of Boolean elements 0 and 1. Then the space of n-tuples of 0's and 1's is the Cartesian product $Z_2 x Z_2 x \cdots x Z_2$ of n Z_2's. This Cartesian product, representing the n-cube, is designated by Z_2^n. Then a Boolean function is a mapping of Z_2^n into Z_2, $Z_2^n \rightarrow Z_2$. The elements of Z_2^n (i.e., n-tuples of 0's and 1's) which map to the element 1 of Z_2 are called 0-cubes or ON-vertices. Two 0-cubes are said to form a 1-cube if they differ in only one coordinate. Thus, the two 0-cubes 101 and 111 form a 1-cube represented as 1x1, where x means that the entry in the second coordinate (also called a component) can be either a 1 or 0. The 0-cubes which form a 1-cube are called faces of the 1-cube. The x indicates a free component, and the others are called bound. The space of all 0-cubes is denoted by K^0, and the space of all 1-cubes is denoted by K^1. Two 1-cubes of K^1 form a 2-cube if the free component is in the same coordinate for both 1-cubes and if exactly one bound component disagrees. The 1-cubes which form a 2-cube are called opposite faces of the 2-cube. The space of all 2-cubes is denoted by K^2. This process continues inductively to give K^r, the space of all r-cubes, $0 \leq r \leq n$.

The operations of obtaining the faces of an r-cube and obtaining an r+1-cube from K^r can be formalized as follows:

Let $(a_1, a_2, \cdots, a_n) = c^r$ be an r-cube, $a_i = 0, 1$ or x, and there are r x's. Then we can find the faces of c^r with the *i-th face operator.*

$$\partial_i^p(a_1, a_2, \cdots, a_n) = \begin{cases} (a_i, \cdots, a_{i-1}, p, a_{i+1}, \cdots, a_n) \text{ if } a_i = x \\ \phi \text{ if } a_i \neq x \end{cases}$$

where p = 0 or 1. We can find a cube c^{r+1} with the *i-th coface operator*

$$\delta_i(a_1, a_2, \cdots, a_n) = \begin{cases} (a_1, \cdots, a_{i-1}, x, a_{i+1}, \cdots, a_n) = c^{r+1} \text{ if } a_i \neq x \text{ and } c^{r+1} \subseteq K(f), \\ \phi \qquad \text{if } a_i = x \text{ or if } c^{r+1} \not\subseteq K(f). \end{cases}$$

The *cubical complex* of function f, $K(f)$ is defined as the collection $K^0, K^1, \cdots, K^r, \cdots, K^n$ and the face and coface operators. A cubical complex is, therefore, an algebraic system made up of algebraic operations and collections of cubes.

Based on the cubical complexes, some of the operations which are useful in obtaining prime implicants, and essential prime implicants are discussed below.

4.2.1 Star-Product (Consensus Operation)

The star-product of two cubes can be defined through the *-product of coordinates of the cubes. The coordinate *-product is defined by Table 2.

*	0	1	x
0	0	y	0
1	y	1	1
x	0	1	x

Table 2
Coordinate *-Product

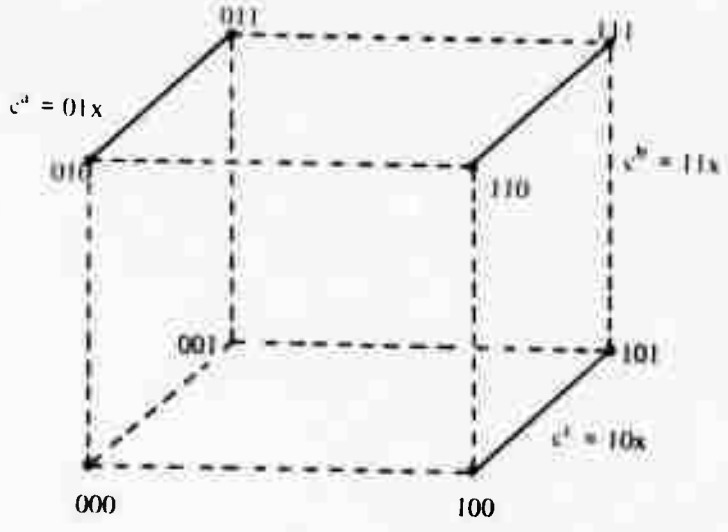
Let $c^r = (a_1, a_2, \ldots, a^n)$ and $c^s = (b_1, b_2, \ldots, b^n)$ be cubes of a complex K. To form the *-product $c^r * c^s$, form $a_i * b_i$ for $1 \leqslant i \leqslant n$. If $a_i * o_i = y$ for more than one i, $c^r * c^s = \phi$, if at most one y appears then

$$c^r * c^s = (m(a_1 * b_1), m(a_2 * b_2), \ldots, m(a_n * b_n)),$$

where $m(a_i * b_i)$ is

$$m(0) = 0,$$
$$m(1) = 1,$$
$$m(x) = m(y) = x.$$

Figure 3 gives several illustrations of the *-product.



$c^d = c^a * c^b = x1x$

$c^e = c^a * c^c = \phi$

$c^f = c^b * c^c = 1xx$

$c^g = (c^a * c^b) * c^c = 1xx$

$c^h = c^a * (c^b * c^c) = x1x$

$c^i = c^b * (c^a * c^c) = \phi$

Illustrations for *-Product of Cubes
Figure 3

The *-product $c^r * c^s$ geometrically is the largest cube c^t (a t-cube) which has opposite $(t-1)$-faces in c^r and c^s, respectively. If the value of $c^r * c^s$ is ϕ, no such c^t exists. The *-product thus has the potentiality of finding a new cube c^t which falls between cubes c^r and c^s, or which may include c^r or c^s. Several properties of the *-product are listed in the following.

1. $c^r * c^s = c^s * c^r$ commutative

2. $c^a * (c^b * c^c) \neq (c^a * c^b) * c^c$ nonassociative

3. If $c^r \subseteq c^s$, then $c^r * c^s = c^r$.

4. If $c^r * c^s = c^t$ and $c^r \subseteq c^t$, then $\delta_i c^r \neq \phi$, for some i, some $\partial_i{}^p c^t \neq \phi$, some sequence of δ_i operations on c^r will produce c^t, and some sequence $\partial_i p$ operations on c^t will produce c^r.

5. If c^{r1} and c^{r2} are opposite faces of some cube c^{r+1}, then $c^{r1} * c^{r2} = c^{r+1} = \delta_i(c^{r1})$, where i is the coordinate in which c^{r1} and c^{r2} differ.

4.2.2 Sharp Operation (#-operation)

This operation is a sort of subtraction operation, and will be described for any two cubes. Note that if we form a#b, where a and b are cubes, we obtain the set of subcubes of a which is not included in b. To give an algebraic definition we first define a coordinate #-operation as given in Table 3.

a_i \ b_i	0	1	x
0	#	y	x
1	y	z	z
x	1	0	z

Table 3

Coordinate #-Operation $a_i \# b_i$

Note that this operation is non-commutative (that is, $a_i \# b_i \neq b_i \# a_i$).

To describe the #-operation between two cubes, let

$$c^r = (a_1, a_2, \cdots, a_n) \text{ and } c^s = (b_1, b_2, \cdots, b_n).$$

$$c^r \# c^s = \begin{cases} c^r \text{ if } a_i \# b_i = y \text{ for any } i \\ \phi \text{ if } a_i \# b_i = z \text{ for all } i \\ \bigcup_i (a_1, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n), \text{ where} \\ a_i \# b_i = a_i = 0 \text{ or } 1 \text{ and the union runs} \\ \text{over all such } i. \end{cases}$$

for example, $xxx \# 11x = 0xx \cup x0x$.

The following properties of the #-operation follow immediately from the definition.

1. $c^r \# c^s = c^r$ if $c^r \cap c^s = \phi$
2. $c^r \# c^s \subseteq c^r$.
3. $c^r \# c^s \neq c^s \# c^r$ non-commutative.
4. $(c^r \# c^s) \# c^t \neq c^r \# (c^s \# c^t)$ non-associative.

The #-operation can also be shown to satisfy the distributive laws:

5. $(c^r \cup c^s) \# c^t = (c^r \# c^t) \cup (c^s \# c^t)$, and
6. $(c^r \cap c^s) \# c^t = (c^r \# c^t) \cap (c^s \# c^t)$,

and also the following type of commutative law:

7. $(c^r \# c^s) \# c^t = (c^r \# c^t) \# c^s$.

4.3 PRIME IMPLICANTS OF A BOOLEAN FUNCTION

A prime implicant α of a Boolean function f is a product which has the following properties:

1. α implies f (i.e., α is an implicant of f)
2. If β is obtained by deleting any literal from α then β does not imply f.

In terms of cubical complex, a cube z of a complex K is a prime implicant of the complex K if there exists no other cube in K which includes z.

Quine-McCluskey tabular method[12, 13] and iterative consensus method[18, 19] are the two most commonly known methods of obtaining the complete set of prime implicants of the given function. These two methods generally require large amounts of computation.

Since only an approximate minimum expression is to be obtained, a complete set of prime implicants is not required. Rather, prime implicants of major concern are those obtainable through enlargement of certain cubes.

4.3.1 Quine-McCluskey Tabular Method

This method is based principally on the theorem $XY + XY' = X$. The first step in this method is to transform the expression (to be simplified) into the canonical sum form. The preceding theorem is then applied exhaustively to obtain all irreducible terms, that is, terms to which the theorem cannot be further applied.

The theorem is applied first to all possible pairs of terms. Two terms to which the theorem can be applied will reduce to one term with one less literal. For example, $ABC + ABC' = AB$. Next, all terms reduced by one literal are examined to see whether they can be combined further, by the application of the theorem, into a term with still fewer literals. This procedure is continued until no further terms can be combined. The resulting irreducible terms are called "prime implicants."

This procedure is more conveniently carried out with binary representations of product terms (called binary terms) than with the algebraic expression itself. The first step is to construct a table by writing down binary terms of the function. These binary terms are grouped according to the number of 1's contained, and the groups are arranged consecutively in the order of increasing number of 1's. Arranging terms in this way, it is necessary only to compare terms in one group with terms in an adjacent group (i.e., the group containing terms with one more 1) in order to apply the theorem $XY + XY' = X$. The terms that can be combined are "checked off" signifying that they are not prime implicants, and the new terms obtained through combination are written down in a new table. When the comparison between terms of every two consecutive groups in the first table is finished, unchecked terms are corresponding to prime implicants.

Terms in the new table are then compared for further combinations. A "–" (indicating a missing literal) in a term must match with a"–" in another term for them to be combined. New terms are arranged into a new table again, and the process is repeated until no new table is created. At the end of the process, unchecked terms from all tables correspond to all prime implicants of the function.

4.3.2 Iterative Consensus Method

To apply the iterative consensus method, the given Boolean expression must be in the sum-of-products form. The theorem $XY + X'Z = XY + X'Z + YZ$ and $X + XY = X$ form the basis for this method. Let $P = xy_1y_2\cdots y_m$ and $Q = x'z_1z_2\cdots z_n$, where it is possible that $y_i = z_j$ for some i and j. The *consensus* of P and Q (i.e., the star product of cubes corresponding to P and Q), also written as $P * Q$, is defined to be $y_1y_2\cdots y_mz_1z_2\cdots z_n$ (with any repeated literals removed) unless $y_i = z'_j$, in which case the consensus is said not to exist. Quine[19] showed that successive additions of consensus terms to a sum-of-products expression and the removed of terms which are included in the other terms (by $X + XY = X$) will result in a *complete sum* which is the sum of all prime implicants.

This procedure can also be implemented as a tabular method using binary terms. The method involves the following steps:

1. Each term of the table is compared with each term above it in the table.

2. If any term is found to be included in another term, the included term is removed from the table.

3. If any two terms have a consensus, the consensus term is compared with all other terms of the table and then added at the bottom of the table if it is not included in any other terms.

4. The process terminates when every term has been compared with all terms lower down in the table. The terms which remain in the table correspond to all the prime implicants.

4.3.3 Cube Enlargement Method

Although the Quine-McCluskey tabular method is straight forward and can be easily implemented, it assumes the given expression to be in canonical sum form. When the canonical expression contains numerous terms, the size of the table becomes inconveniently big, and the process becomes a time consuming computation. The iterative consensus method requires only that the expression is in sum-of-products form. Therefore, it generally has a smaller table to begin, but the process is more involved. Both methods calculate the whole set of prime implicants. For a certain class of functions the number of all prime implicants may be very big. For example, consider a function of 3k variables with a cubical complex containing all cubes having exactly k coordinates equal to 1, k coordinates equal to 0, and k coordinates equal to x. It is easy to see that each of these cubes corresponds to a prime implicant of the function. Therefore, for this special class of functions the number of prime implicants is

$$\binom{3k}{k}\binom{2k}{k} = \frac{(3k)!}{(k!)^3}$$

which is 1680 for k = 3. Consequently for such functions both the Quine-McCluskey tabular method and the iterative consensus method are very inefficient.

Quite often it may not be necessary to obtain a minimum form, either if the minimization process becomes tedious and time consuming or if the cost of the circuit is not of primary concern. Hence an algorithm of finding an approximate minimum based on the cube enlargement is developed.

In the cube enlargement method, the given function must be expanded into a sum-of-products form first as in the iterative consensus method. The method will be described in terms of cubical complex, and don't cares will be considered also. This method is a modification of Miller's Local Extraction Algorithm[16].

Let C_o be the set of all ON-cubes (corresponding to products in the expanded form), and K_o be the cubical complex of C_o. Also let D_o be the set of all don't care cubes (corresponding to products of the don't care function), and N_o be the cubical complex of D_o.

The procedure involves the following steps:

1. All cubes of C_o are arranged in the order of increasing number of x in each cube.

2. Let $c^r = (c_1, c_2, \text{----}, c_n)$ be a cube of C_o, and c_i be the first non-x component. Form $z = (c_1, c_2, \text{----}, c_{i-1}, x, c_{i+1}, \text{----}, c_n)$ and $\bar{c}^r = (c_1, c_2, \text{----}, c_{i-1}, \alpha, c_{i+1}, \text{----}, c_n)$, where $\alpha = 0$ if $c_i = 1$ and $\alpha = 1$ if $c_i = 0$. Then check to see if z is in the cubical complex $k_1 = K_o \cup N_o$. This can be done by using the #-operation. Thus, $c^r \# ((C_o \; c^r) \{ \cdot D_o) = \phi$ (empty), if and only if z is in K_1. If z is in K_1, c^r is replaced by z and the operation is repeated on the next non-x coordinate of z. If z is not in K_1, we try the same operation on the next succeeding non-x coordinate of c^r.

This process of expanding c^r is continued until all non-x coordinates have been tested. The resultant cube is the prime implicant enlarged from c^r. After applying the second step to each c^r of C_o, the prime implicant cover C_1 of K_1 is obtained. The number of prime implicants obtained is less than or equal to the number of terms in the original sum-of-products expression.

4.4 APPROXIMATE MINIMUM COVER

A minimum expression is the sum of an irredundant set of prime implicants such that one of the minimality criteria is satisfied. To find an absolute minimum expression, one has to find the whole set of prime implicants which in some cases is terribly big. To find an approximate minimum, we need only the set of prime implicants obtained by cube enlargement. The procedure of obtaining an approximate minimum cover from this set of prime implicants is as follows:

Let C_1 be the set of prime implicants obtained by means of the cube enlargement method. We wish to find a minimum cover which is a minimum subset of prime implicants in C_1 such that every vertex in K_o is covered by some prime implicants in the subset.

The first part is to find if any cube c of C_1 covers some vertex in K_o which is not covered by any other cube in C_1. A cube c is of this type if

$$c \# D_o \# (C_1 \; c) \neq \phi \qquad (4.4.1)$$

such a cube is called a *pseudo essential prime implicant*, and it must be included in the approximate minimum cover. The set of all pseudo essential prime implicants will be denoted by F_1.

The second part is to find the approximate minimum cover by iteratively choosing cubes in $C_1 - F_1$. In selecting cubes from $C_1 \quad F_1$, bigger cubes with larger number of vertices not cover by F_1 and D_o are

preferential. This is because a bigger cube has fewer literals, and vertices not covered by F_1 and D_o are ON-vertices which must be covered in the minimum cover. This preference in selection, can be accomplished by ordering cubes in $C_1 - F_1$ according to their *cost factors*. The cost factor of a cube q is defined as

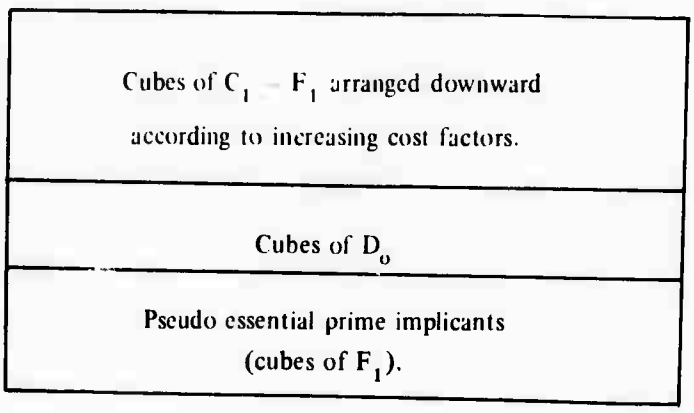
$$\beta = 2^r + k \cdot |a|$$

where 2^r is the number of vertices in cube q (r is the dimension of cube q),
k is a weight,

and $|a|$ is the number of vertices in a, which is the set of q's vertices not covered by F_1 and D_o, namely

$$a = q \# D_o \# F_1 \qquad (4.4.2)$$

To facilitate the selection, cubes in $C_1 - F_1$ are arranged in the order of increasing cost factors. This list of cubes is then followed by the list of cubes in D_o and the list of cubes in F_1 to form a linear array as shown in Figure 4.



Arrangement of Cubes for Obtaining an Approximate Minimum Cover

Figure 4

The selection algorithm is as follows:

1. Starting from the first cube in the array, each cube in the first list is #-operated by every cube below it in the whole array.

2. Whenever an empty result is obtained in a chain of #-operations, the cube under consideration is discarded and then pick up the cube immediately below it for consideration.

3. If what results from the complete chain of #-operations is non-empty, we must select the cube under consideration and move it to the bottom of the third list.

4. The process terminates when the last cube in the first list has been considered for selection.

When the process stops, the set of cubes in the third list corresponds to the approximate minimum cover.

The designer can choose a minimum cover with fewer literals or fewer terms by adjusting the weight k. In general, if a design with fewer terms is preferred, a higher weight should be used. Conversely, if a design with fewer literals is preferred, a lower weight should be used. Weights of ½, 1, and 2 are used in the implementation.

REMARK

Two chains of #-operations are carried out during the minimization as implied by equations (4.4.1) and (4.4.2). They are rewritten in the following:

$$c \# D_o \# (C_1 - c) = \phi \qquad (4.4.1)$$

$$q \# D_o \# F_1 = a \qquad (4.4.2)$$

where c is in C_1 and q is in $C_1 - F_1$.

Since a cube in $C_1 - F_1$ is also in C_1, the computation of $(q \# D_o)$ can be saved if the intermediate results $(c \# D_o)$'s are stored during the computation of equation (4.4.1). In general, however, because $c \# D_o$ is not a single cube, a large memory is required to store all $(c \# D_o)$'s.

4.5 PROGRAM IMPLEMENTATION

Since the LINC is a binary computer, each bit can only represent either 0 or 1. But a variable in a product may be true, not, or missing. Hence two memory words (or bytes) are required to represent a product (i.e., a cube). For example, in the case of three variable functions, the product AC'(i.e., 1x0) is represented by the two binary words 100 and 010. In the first word a 1 represents a 1, while a 0 represents either a 0 or an x in the binary representation of the product. In the second word a 1 represents x, while a 0 represents either a 1 or a 0. The first word is denoted by N and the second word by P in the flow chart of the minimization program given in Appendix 7.2. This internal representation is called the N-P representation. As the first step of the minimization procedure, product term is transformed into its N-P representation. After the minimum cover is obtained, the N-P representation of each cube in the cover is transformed into a product term.

4.6 EXAMPLE

Consider the Boolean function F with F_c and F_d given as follows:

$$F_c(H,G,E,A,B,C,D) = H'AB'C'D' + H'E'A'BC'D' + H'G'E'A'BC'D' + H'G'A'B'CD' + H'A'B'C'D' + G'E'ABD' + G'EBCD + H'G'E'BD + H'G'A'BCD$$

$$F_d(H,G,E,A,B,C,D) = H'E'A'BC'D + H'GA'B'CD$$

The prime implicants obtained through the cube enlargement are

H'E'A'BC'
H'E'A'C'D'
H'A'B'D'
H'G'A'BCD
H'B'C'D'
G'E'ABD
G'EBCD
H'G'E'BD.

The approximate minimum cover obtained is

H'E'A'BC'
H'A'B'D'
H'B'C'D'
G'E'ABD
G'EBCD
H'G'E'BD

where the first five are pseudo essential prime implicants. The approximate minimum expression is then

$$H'E'A'BC' + H'A'B'D' + H'B'C'D' + G'E'ABD + G'EBCD + H'G'E'BD.$$

4.7 CONCLUSION

Many Boolean expression minimization algorithms have been developed in the past, almost all of them are for obtaining absolute minimum expressions. Due to the following reasons, however, it is sometimes not preferred to obtain the absolute minimum expression.

1. The computation is time consuming and requires large computer memory.

2. The switching circuit realizing the absolute minimum expression is sometimes the most unreliable circuit

3. Redundancy is sometimes added to increase reliability and check out capabilities.

Therefore, it is often more preferable to compute an approximate minimum expression than an absolute minimum expression. For this reason, the algorithm for obtaining an approximate minimum expression is developed.

The algorithms presented in Sections 4.3.3 and 4.4 have been implemented on the LINC computer. These algorithms, however, are useful in any binary machine. In principle, there is no limitation to the number of variables of a Boolean function.

5. COMPUTER EXPANSION OF BOOLEAN EXPRESSIONS

5.1 INTRODUCTION

In computer minimization of Boolean expressions, it is generally assumed that Boolean expressions are given in canonical form (i.e., sum-of-minterms form), normal form (i.e., sum-of-products form), or their equivalents. Since in practice Boolean expressions as they are originally given are very often not in the assumed form, it is necessary to preprocess them by expanding them into the assumed form. Although relatively easy and straight forward, this process could be troublesome and become a source of error if performed manually. Therefore, this process is an essential step in the automation of Boolean expression minimization.

Minimization methods can be classified into two categories according to whether the canonical form expansion is required [12-14] or simply a normal form expansion is required[15-18]. Generally the amount of computation and memory needed in the minimization process is proportional to the number of terms in the expanded expression. Therefore methods of the second category are more suitable to computer processing where the normal form expression obtained has considerably fewer terms than the canonical form expression has. This is very often the case if the normal form expression is obtained by applying DeMorgan's theorems, associative laws, and distributive laws. Accordingly the development of efficient computer algorithms to perform this normal form expansion is important.

Computer algorithms are presented in section 5.2 for applying DeMorgan's theorems algebraically to any Boolean expression containing "AND," "OR," and "NOT" operators. The resulting expression can then be expanded algebraically into a normal form expression by applying associative laws and distributive laws, using the algorithm of section 5.3. Main features of these algorithms are:

1. Each algorithm is a one-pass algorithm.

2. Each algorithm is syntax-oriented.

5.2 ALGORITHMS FOR APPLYING DEMORGAN'S THEOREMS

Boolean expressions given are assumed to be in the most commonly used infix form as defined in subsection 2.3.1. In this form, DeMorgan's theorems can be written as $(X+Y)' = x'\,y'$ and $(XY)' = X' + Y'$.

The Boolean expression is scanned backward from the end (i.e., from right to left), and DeMorgan's theorems are applied whenever applicable. Two different algorithms which perform this process are studied. They act as syntax analyzers and transform the expression almost symbol by symbol in one pass. They are given in Backus Normal Form[20] with output imbedded as shown in algorithms 1 and 2.

Algorithm 1:

<expression> :: = <term>{ + ["+"] <term>}

<term> :: = <factor> {<factor>}

<factor> :: = <variable> [the same variable] /

'<variable> ["'" and the same variable] /

) ["("] <expression> (["("] /

') <c expression> (

<c expression> :: = { <single variable> / [")"] <c term> ["("] }

{ + { <single variable> / [")"] <c term> ["("]}}

<c term> :: = <c factor> { ["+" if the current symbol is not + or ( ] <c factor> }

<c factor> :: = <variable> ["'" and the same variable] /

' <variable> [the same variable] /

) <c expression> (/

') <expression> (

<single variable>* :: = <variable> { + / (}** ["'" and the same variable]

' <variable> { + / (}** [the same variable]

<variable> :: = A/B/C/ . . . . . /Z

*A false return from the <single variable> routine causes the input string pointer to move back to the place it has upon entering this routine.

**A successful recognition of + or (at this point does not cause advancement of the input string pointer. This is the only exception to the general rule of advancing the input string pointer.

Several metalinguistic symbols are used here and in the following algorithms. < > denotes a syntactic class. ::= means "is defined as," and / has the meaning of "OR." $\{\ \}$ denotes iteration, i.e., zero or more concatenated occurrences of the contents of the brackets. $(\ )$ denotes ordered alternation, i.e., the first alternation is taken first if applicable, otherwise take the second alternation, etc. [ ] denotes symbols to be put out or actions to be taken. In all algorithms, a successful recognition of a symbol causes advancement of the input string pointer by one symbol position (i.e., get next symbol).

When this algorithm is applied to the input expression

$$((A+B)C+D')' + C((A+B)'D)' + ((B+C)'D)'$$

the output expression is

$$(A'B'+C')D + C(A'B+D') + (B+C+D')$$

Redundant parenthesis pairs in the input expression are acceptable, and they will appear in the output expression also. Moreover, as shown in the third term of the output expression, the algorithm generates a type of redundant parenthesis pair even if the input expression is free from redundant parentheses. This type of redundancy is inevitable for a unidirectional-scan one-pass algorithm. This is so because, as one might have noticed by comparing the last two terms of the input and output expressions, the algorithm would not know that there is no other factor in the term when it scans the factor $((B+C)'D)'$ in the third term.

In cases where the generation of more redundant parenthesis pairs is tolerable, a slightly simpler algorithm, given as algorithm 2 in the following, can be used. When algorithm 2 is applied to the same input expression it generates

$$((A')(B')+C')(D) + C((A+B)+D')+((B+C)+D').$$

It is to be noticed that this algorithm produces redundant parentheses around single variable factors. In fact the purpose of having the syntactic class <single variable> in algorithm 1 is mainly to eliminate this kind of redundant parentheses. Though redundant parenthesis pairs are harmless, they not only occupy memory but also make the output expression look clumsy. When this process is followed by the algorithm for applying associative laws and distributive laws to be discussed in the following section, all parentheses will be removed eventually. Excessive number of redundant parentheses, however, will decrease the efficiency of that algorithm greatly.

5.3 AN ALGORITHM FOR APPLYING ASSOCIATIVE LAWS AND DISTRIBUTIVE LAWS

Associative laws are

$$(X + Y) + Z = X + (Y + Z) = X + Y + Z,$$

and

$$(XY)Z = X(YZ) = XYZ.$$

Algorithm 2:

<expression> :: = <term> { + ["+"] <term> }

<term> :: = <factor> {<factor>}

<factor> :: = <variable> [the same variable] /

' <variable> ["'" and the same variable] /

') [")"] <c expression> (["("] /

) [")"] <expression> (["("]

<c expression> :: = <c term> { + ["(" , ")"] <c term>}

<c term> :: = <c factor> { ["+"] <c factor>}

<c factor> :: = <variable> ["'" and the same variable] /

' <variable> [the same variable] /

') [")"] <expression> (["("] /

) [")"] <c expression> (["("]

<variable> :: = A/B/C/D/ . . . . . /Z

Distributive laws are

$$X(Y + Z) = XY + XZ,$$

$$(X + Y)Z = XZ + YZ,$$

and

$$(X + Y)(X + Z) = X + YZ.$$

The expression obtained from applying any algorithm of the previous section is the one to be processed by this algorithm. The expression is scanned unidirectionally from the beginning (i.e., from left to right) in one pass, and one of these laws is applied whenever applicable. The algorithm, given as algorithm 3, analyzes the syntax and processes the expression with the aid of a pushdown stack (called the primary stack) of component stacks (called secondary stacks). The stack structure is shown in Figure 5, and the algorithm is again given in Backus Normal Form with actions imbedded. A cell of the primary stack stores a secondary stack pointer, while a cell of a secondary stack stores an item.

Algorithm 3:

<expression> :: = <term> [output operation]

{ + ["+"] <term> [output operation]}

<term> :: = <factor> [pushdown the primary stack]

{<factor> [1. * operation
2. pushdown the primary stack, if the current symbol is not)]}

<factor> :: = <primitive> / (<p expression>)

<primitive> :: = {<variable> / <variable> '}

{<variable> / <variable> '} [store the primitive as an item to the top secondary stack]

<p expression> :: = <term> { + <term> [1. + operation
2. pushdown the primary stack, if the current symbol is +]}

<variable> :: = A/B/C/ . . . . . /Z

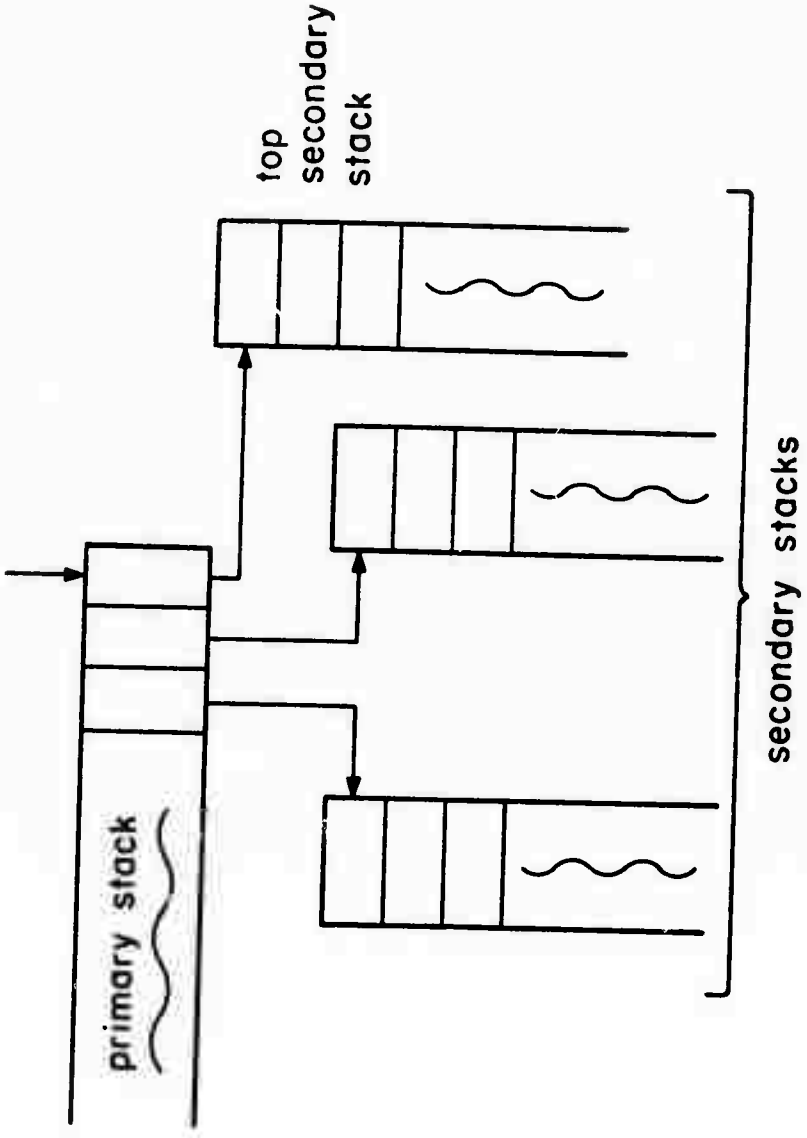


Figure 5
Stack Structure

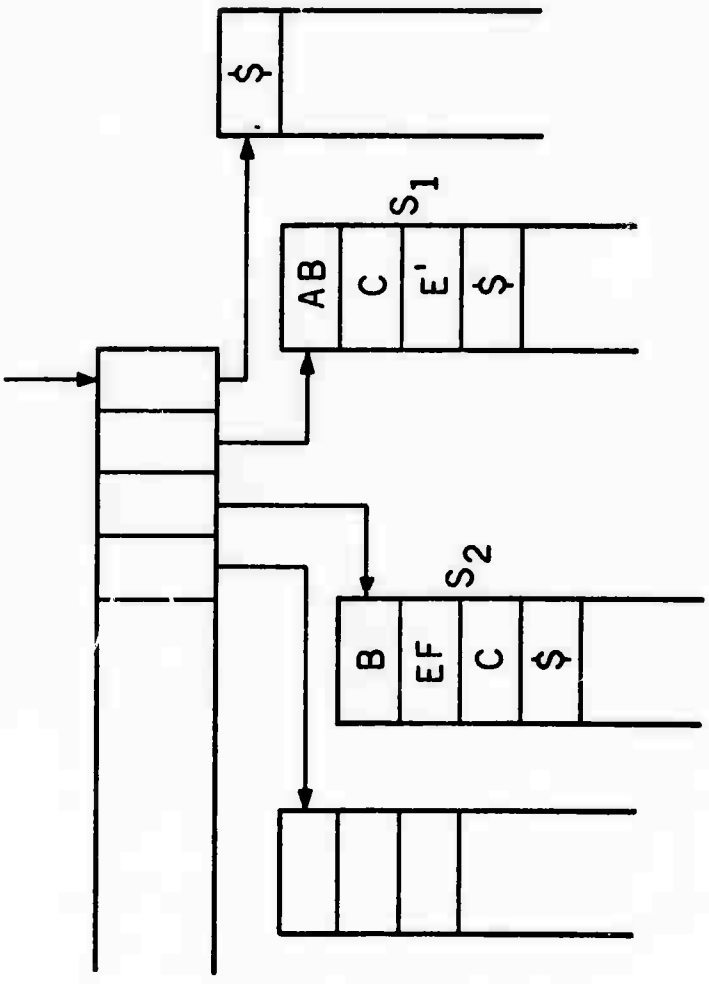


Figure 6 (a)

Figure 6. Changes of stack contents and pointers illustrating the + operation and the * operation. (a) Preceding the + operation or the * operation.
(b) After the + operation: (c) After step 1 of the * operation.
(d) After step 2 of the * operation.

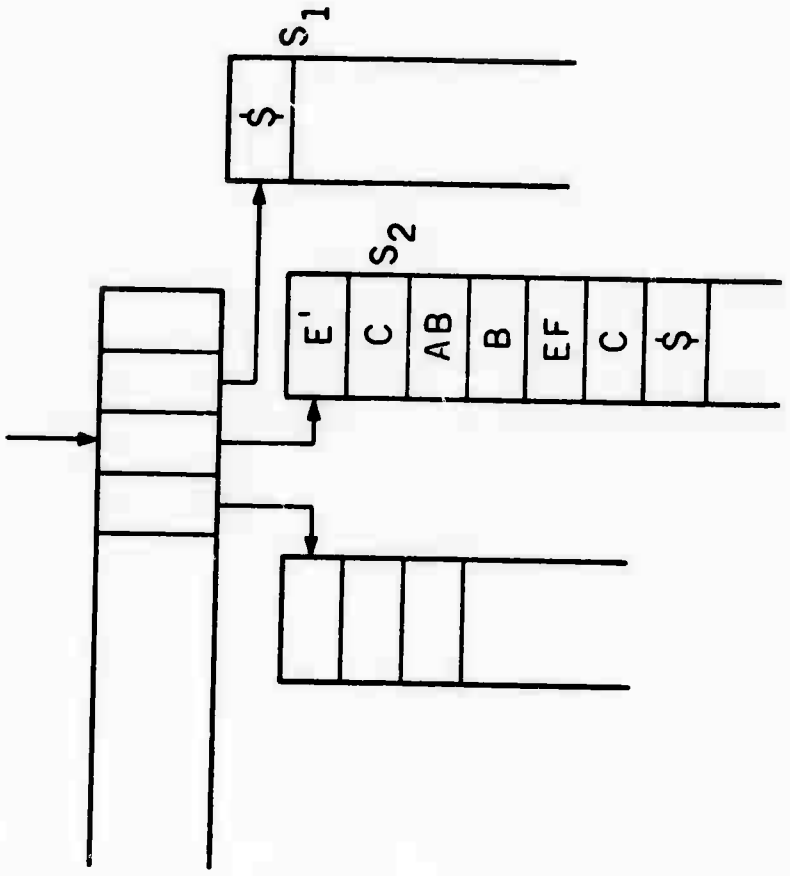


Figure 6(b)

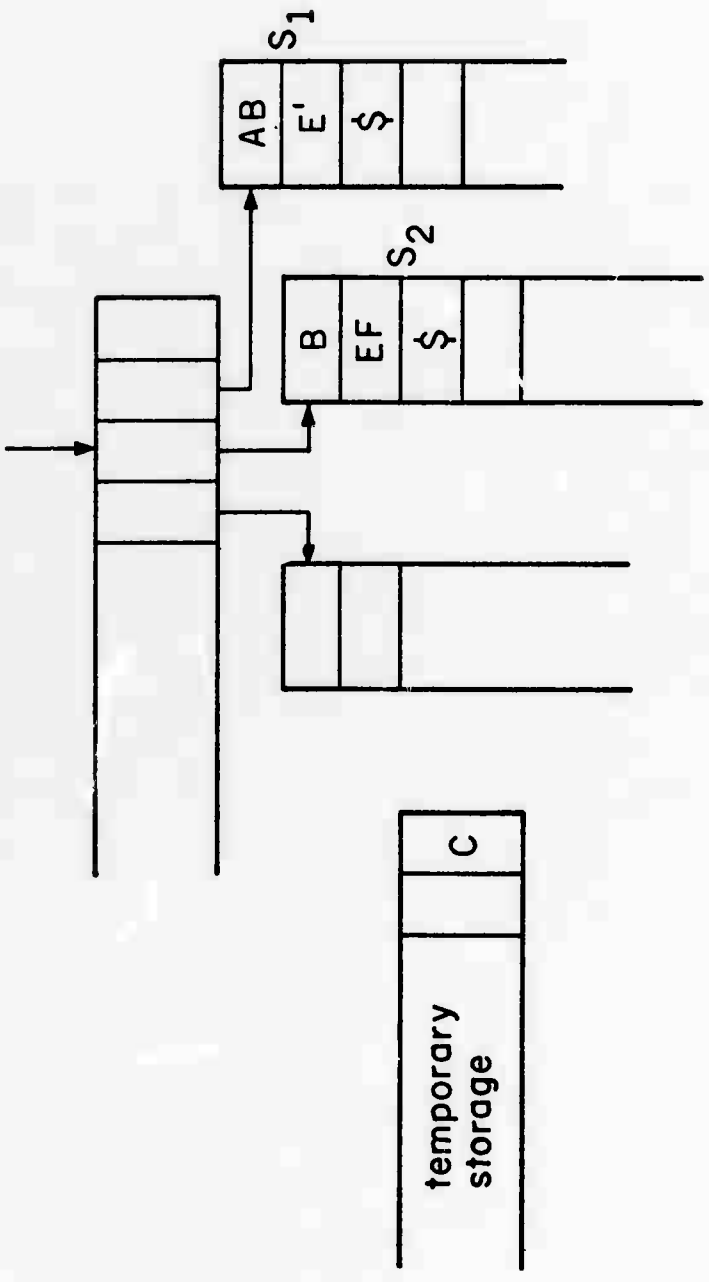


Figure 6(c)

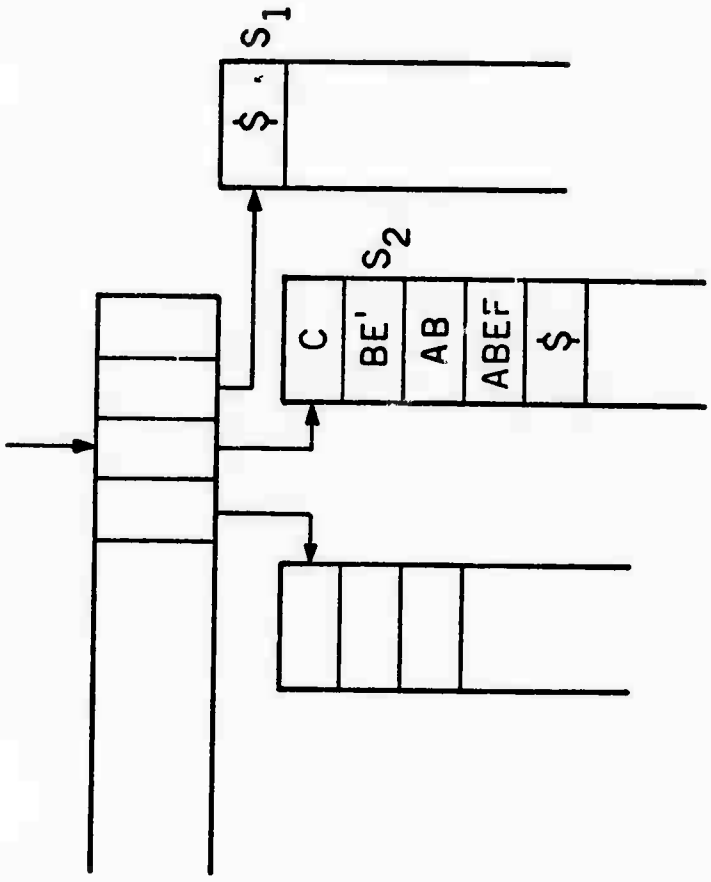


Figure 6(d)

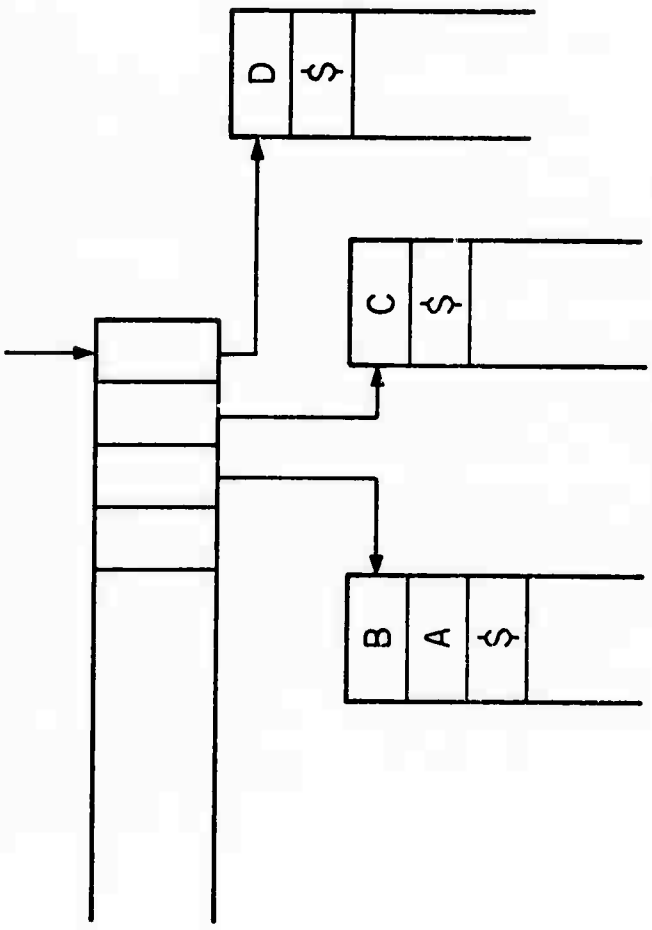


Figure 7(a)

Figure 7. Three key steps in the process of applying algorithm 2 to an expression.

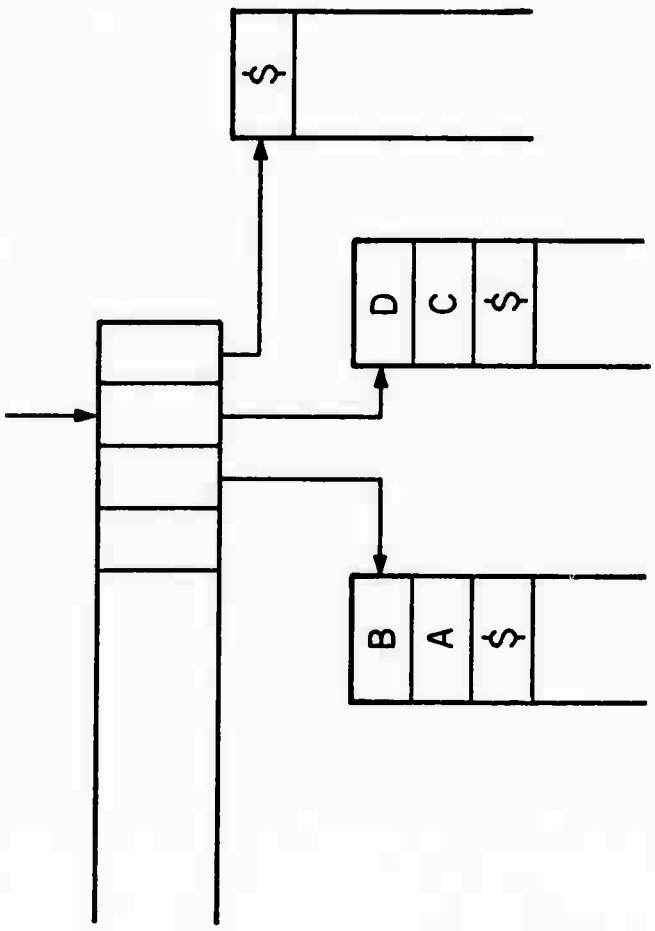


Figure 7(b)

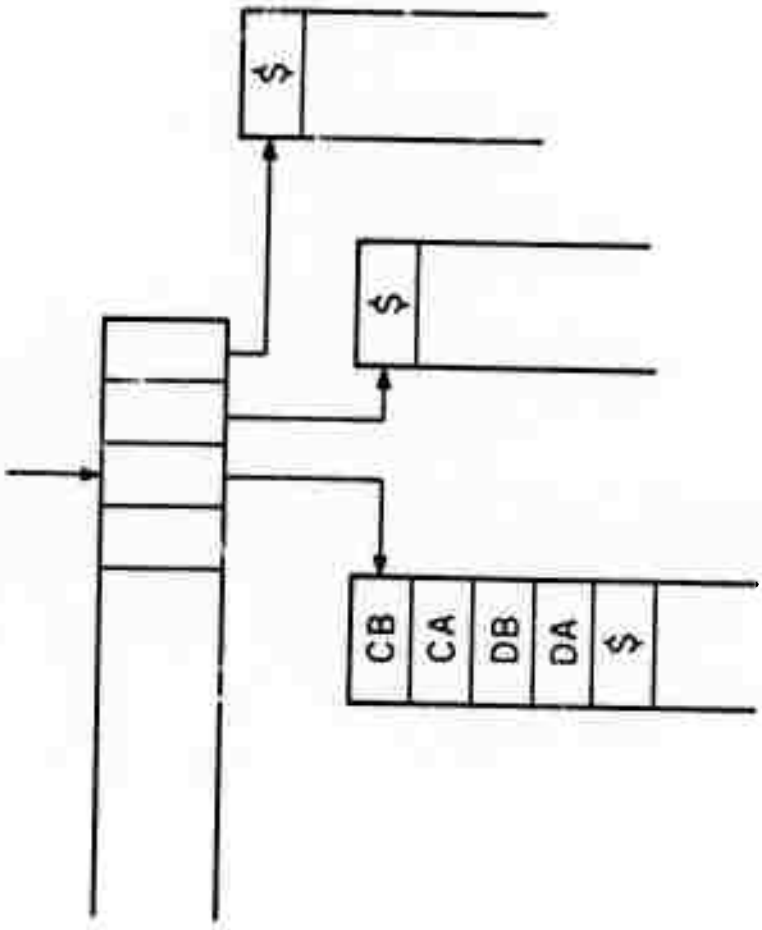


Figure 7(c)

The output operation prints out each item in the top non-empty secondary stack as a product term with a + sign between every two terms. In cases where the cubical cover corresponding to a normal form expression is to be obtained directly, the output operation calls for transformation of these items into their cubical representations and stores them away. The + operation and the * operation both operate on the top two non-empty secondary stacks. Let the top non-empty secondary stack and the one immediately below it be denoted by S_1 and S_2 respectively. The + operation transfers all items in S_1 to S_2, clears S_1 by moving its pointer to the bottom, and then moves the primary stack pointer to S_2. The * operation consists of the following two steps:

1. Delete items common to S_1 and S_2, and record these items in a temporary storage.

2. Distribute each remaining item in S_1 to all remaining items in S_2, transfer all items in the temporary storage to S_2, clear S_1 by moving its pointer to the bottom, and then move the primary stack pointer to S_2. In the distribution process, the multiplicative complementarity law ($XX' = 0$) and the multiplicative idempotence low ($XX = X$) can be applied.

Thus, if the stack contents and pointers at some stage of the process are as shown in Figure 6(a), then they become as shown in Figure 6(b) after the + operation. After step 1 and step 2 of the * operation, they will be as shown in Figure 6(c) and Figure 6(d) respectively.

To help in understanding this algorithm some key steps in the process of applying it to the expression((A+B)(C+D)E+F)G will be explained. After D is recognized as a variable, the input string pointer moves to the next symbol, and D is stored as an item in the top secondary stack. At this stage the stack contents and pointers are shown in Figure 7(a). After D is recognized as a term and the + operation performed, they become as shown in Figure 7(b). Since the symbol pointed by the input string pointer at this time is), it recognized C+D as a p expression and (C+D) as a factor. The * operation is then performed, and the stack contents and pointers become as shown in Figure 7(c).

5.4 PROGRAM IMPLEMENTATION

The flow chart of the Boolean expression expansion program is shown in Appendix 7.3.

5.5 CONCLUSION

Three computer algorithms have been presented; two for applying DeMorgan's theorems and the other for applying associative laws and distributive laws. By using these algorithms any well-formed Boolean expression containing "AND," "OR," and "NOT" operations can be expanded into its normal form in two passes. These recursive algorithms may not be most efficient to achieve their goals, yet they are easy to understand because they are syntax-oriented. They have been programmed and tested on the LINC, and the whole program occupies less than a thousand 12-bit words of memory.

Symbol manipulation languages such as SNOBOL[21] and FORMAC[22] can be used to program the expansion of a Boolean expression with relative ease. Aside from the necessity of a big machine to run a general symbol manipulation language, however, special-purpose algorithms such as these are undoubtedly much more efficient.

6. CONCLUSION

A set of computer oriented algorithms useful in logic design automation has been developed. These algorithms are programmed on the LINC computer as a set of seven routines. The routines can be used independently or jointly. Figure 8 explains how the routines can be linked together to form a computer aided logic design system. In the figure, each oval box represents a routine while each rectangular box indicates the form by which a Boolean function is represented. The double-lined arrows indicate entry points. The system can be entered from four different entry points.

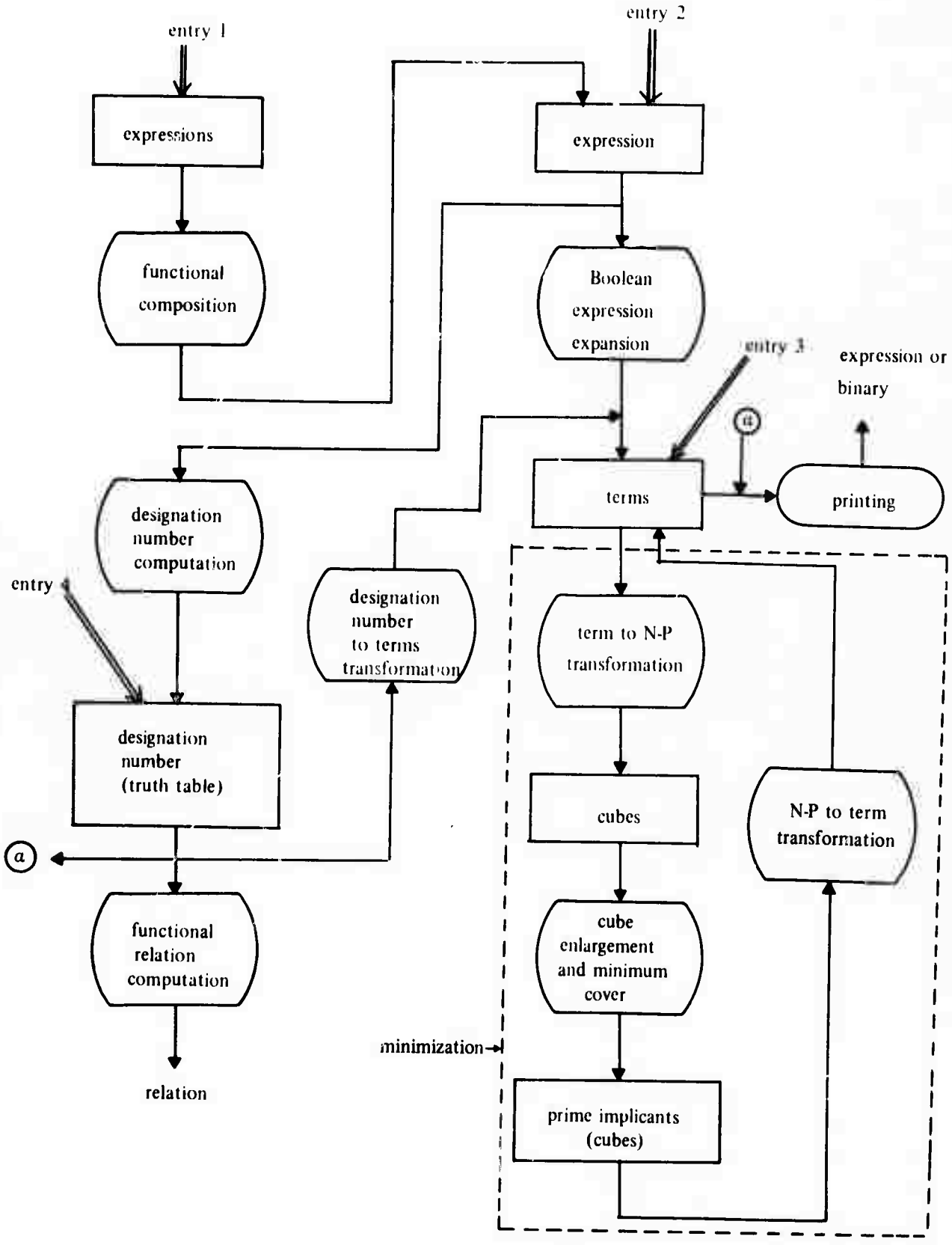
In principle, the application of these algorithms is not limited by the number of variables. During the development, particular attention has been paid to generality, modularity, and machine independence.

Some possible future extensions of this research are:

1. Solution of Boolean equations using designation numbers.

2. Detection of functional equivalence resulting from complementation of variables.

3. Analysis and synthesis of sequential circuits.



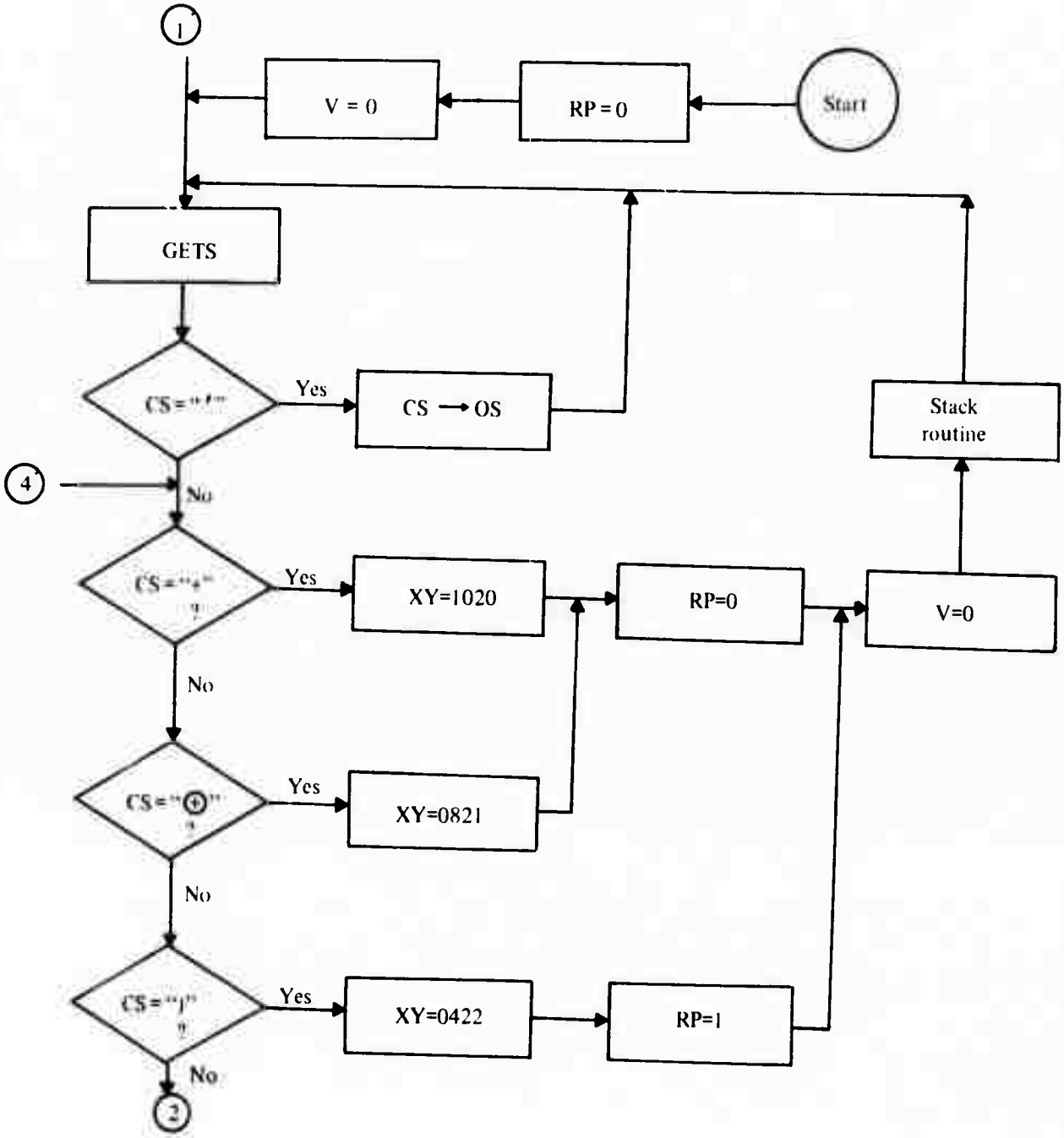
A Computer Aided Logic Design System

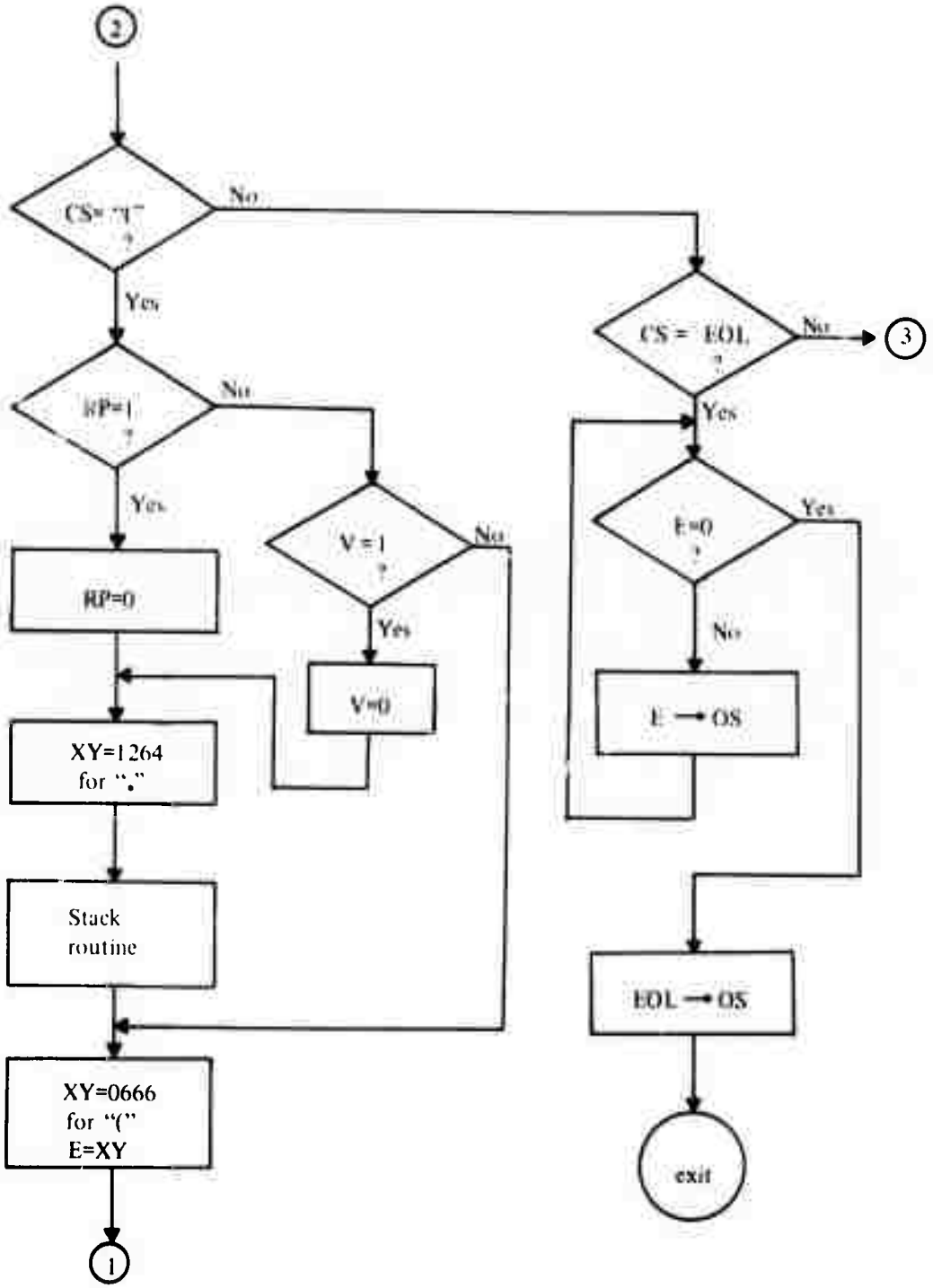
Figure 8

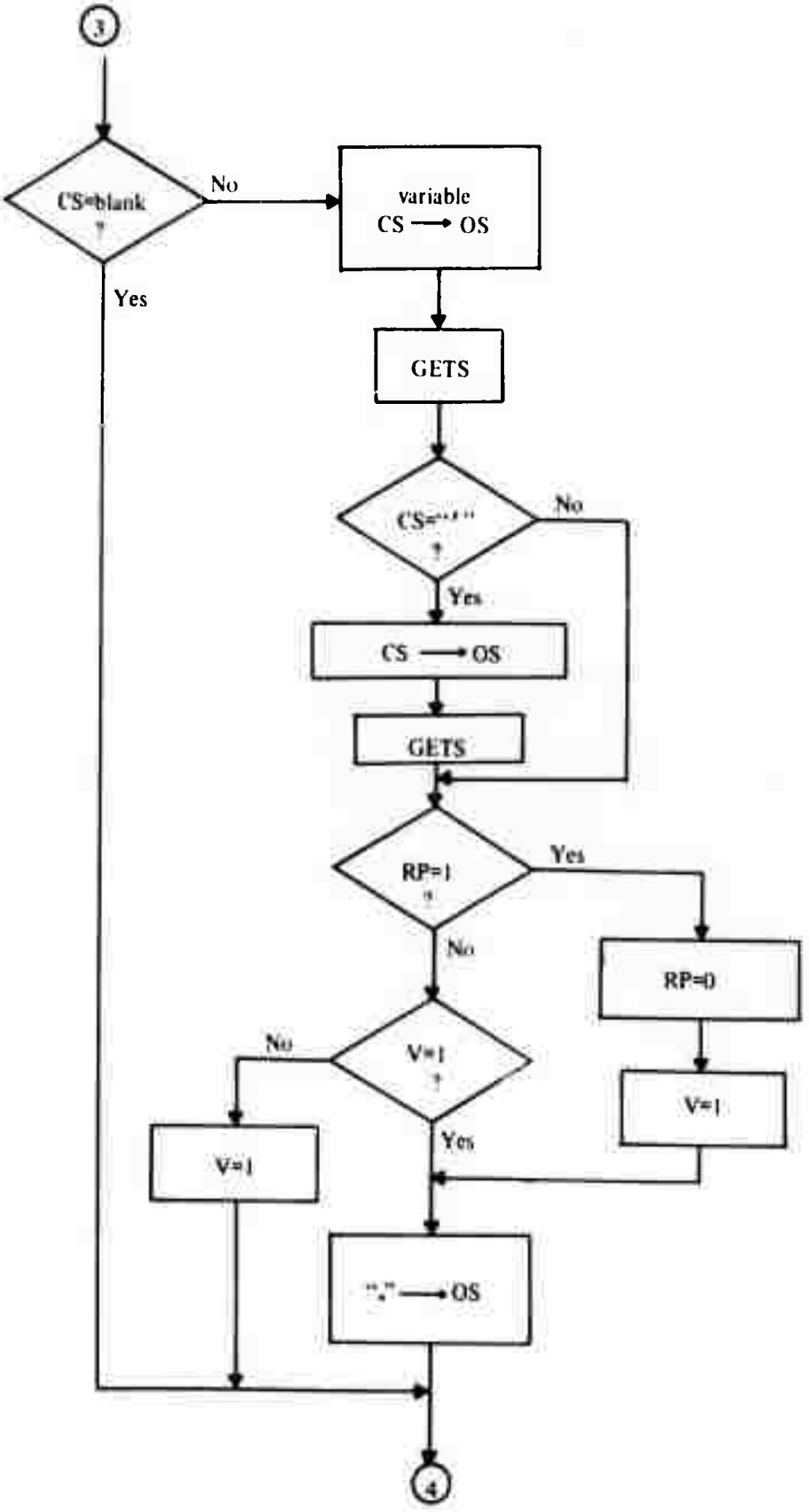
7. APPENDICES

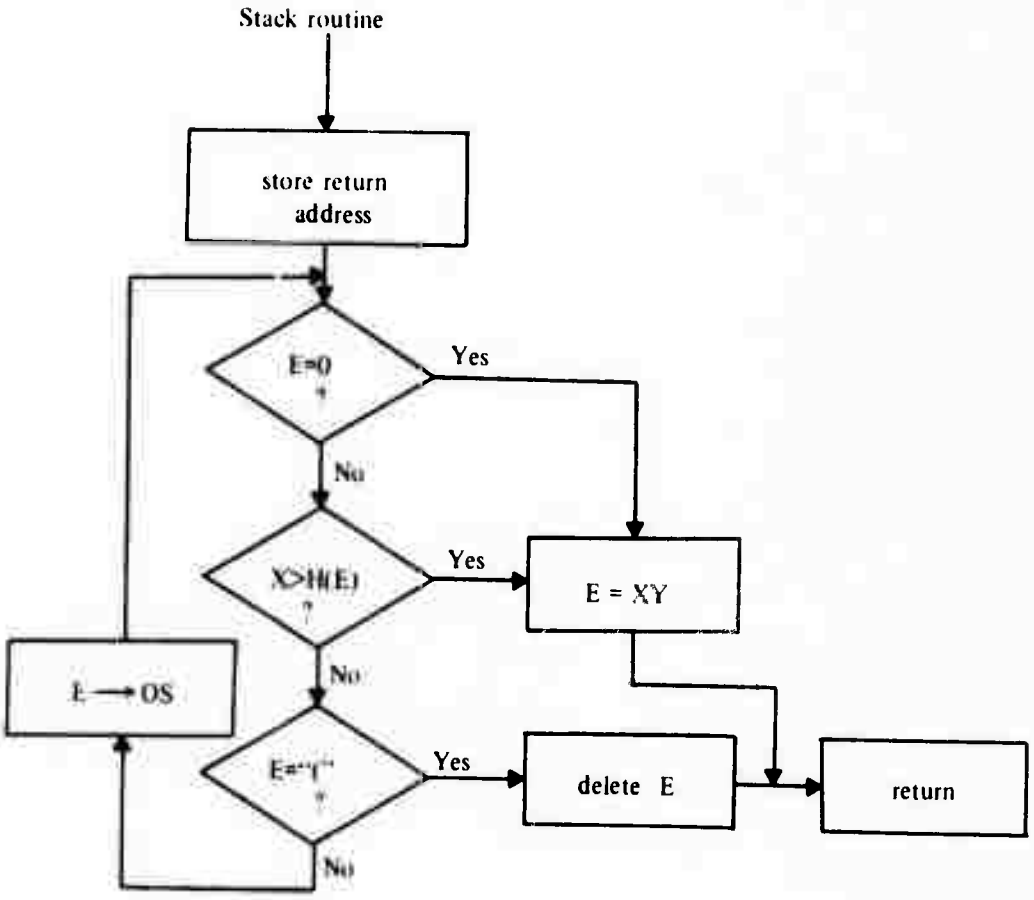
APPENDIX 7.1

The Flow Chart for Transforming a Boolean Expression into the Early Reverse Polish Form




2
CS= "("
?
No
Yes
RP=1
?
No
Yes
RP=0
V=1
?
No
Yes
V=0
XY=1264
for "."
Stack
routine
XY=0666
for "("
E=XY
1
CS = EOL
?
No
3
Yes
E=0
?
Yes
No
E → OS
EOL → OS
exit


3
CS=blank
?
No
variable
CS → OS
Yes
GETS
CS="."
?
No
Yes
CS → OS
GETS
RP=1
?
Yes
No
RP=0
V=1
?
No
V=1
Yes
V=1
"." → OS
4



Notations

CS: Current input symbol

V: Variable flag (set to 1 when CS is a variable and reset to 0 when CS is), +, or ⊕)

RP: Right parenthesis flag (set to 1 when CS is a right parenthesis, and reset to 0 when CS is + or ⊕)

OS: Output string

EOL: End of the expression

XY: Y and X denote an operator and its hierarchy number respectively

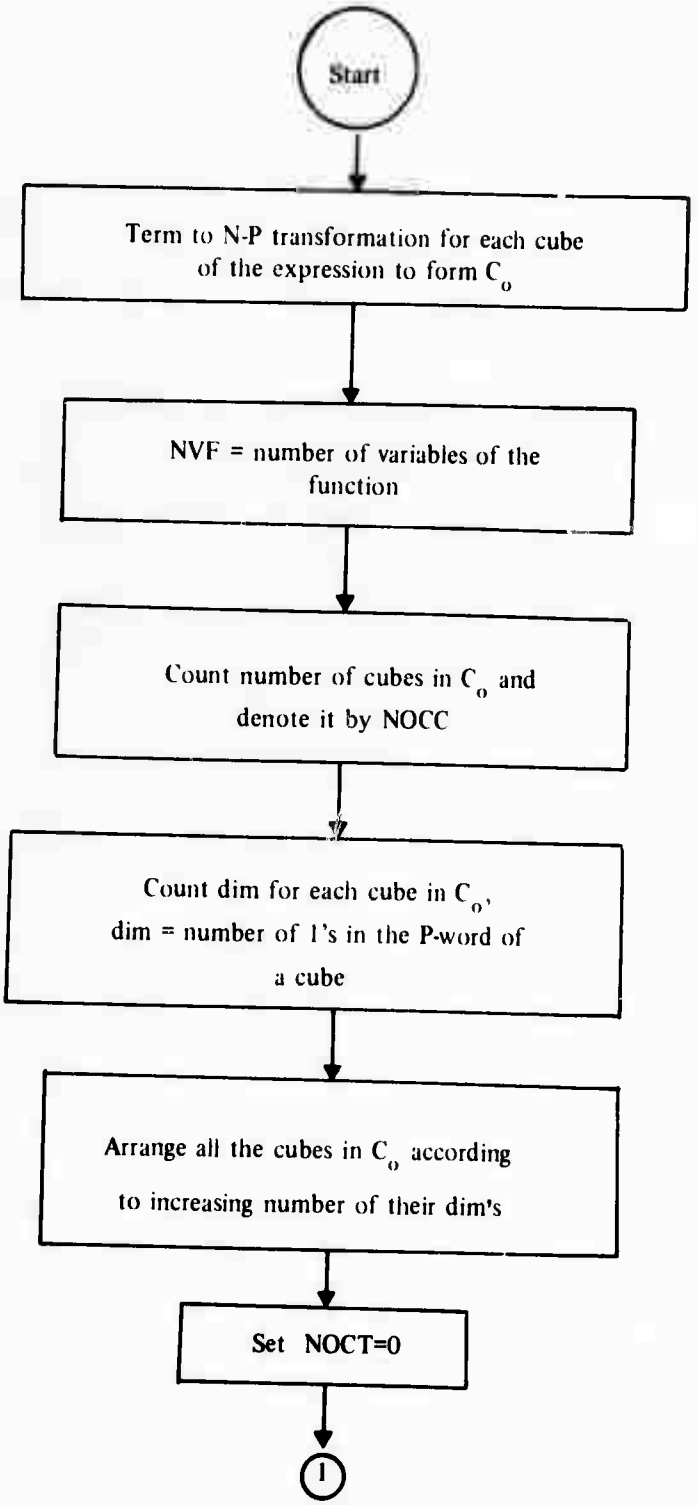
E: Top element of the push-down list N

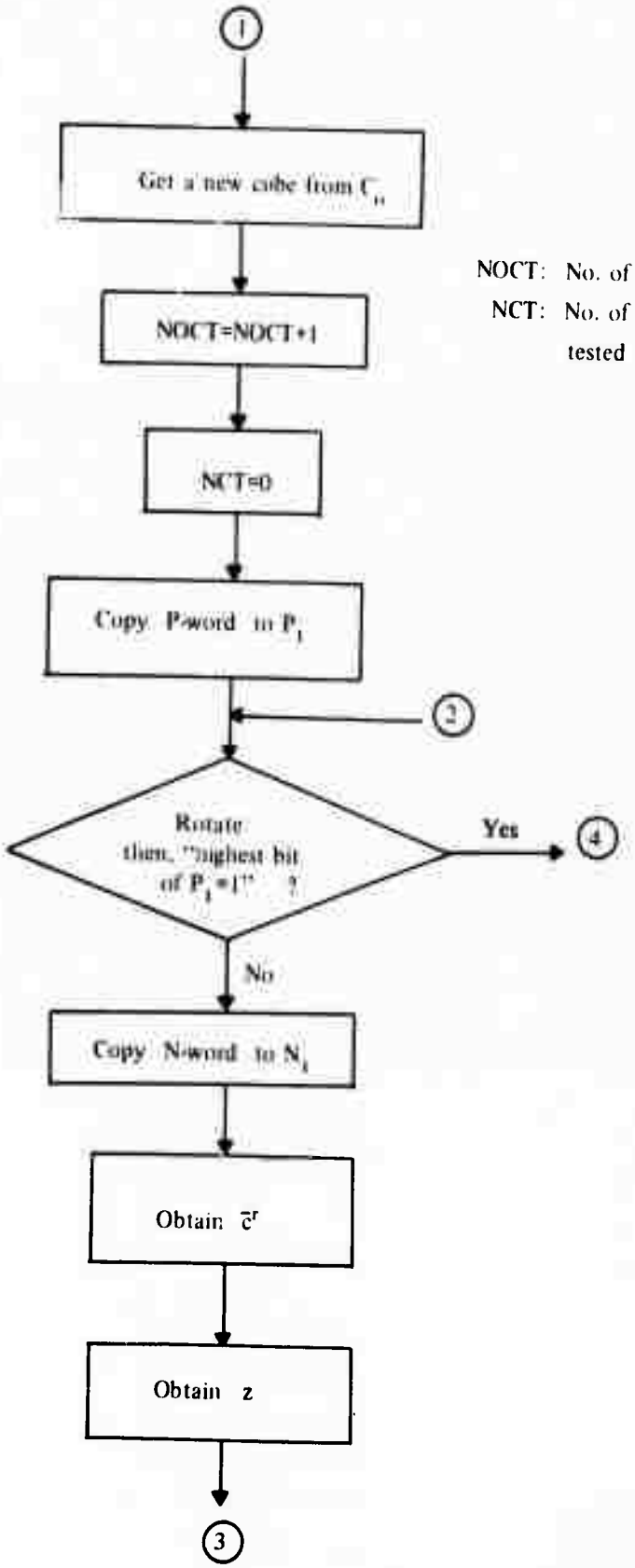
H(E): The hierarchy number of E

APPENDIX 7.2

The Flow Chart for Boolean Expression Minimization

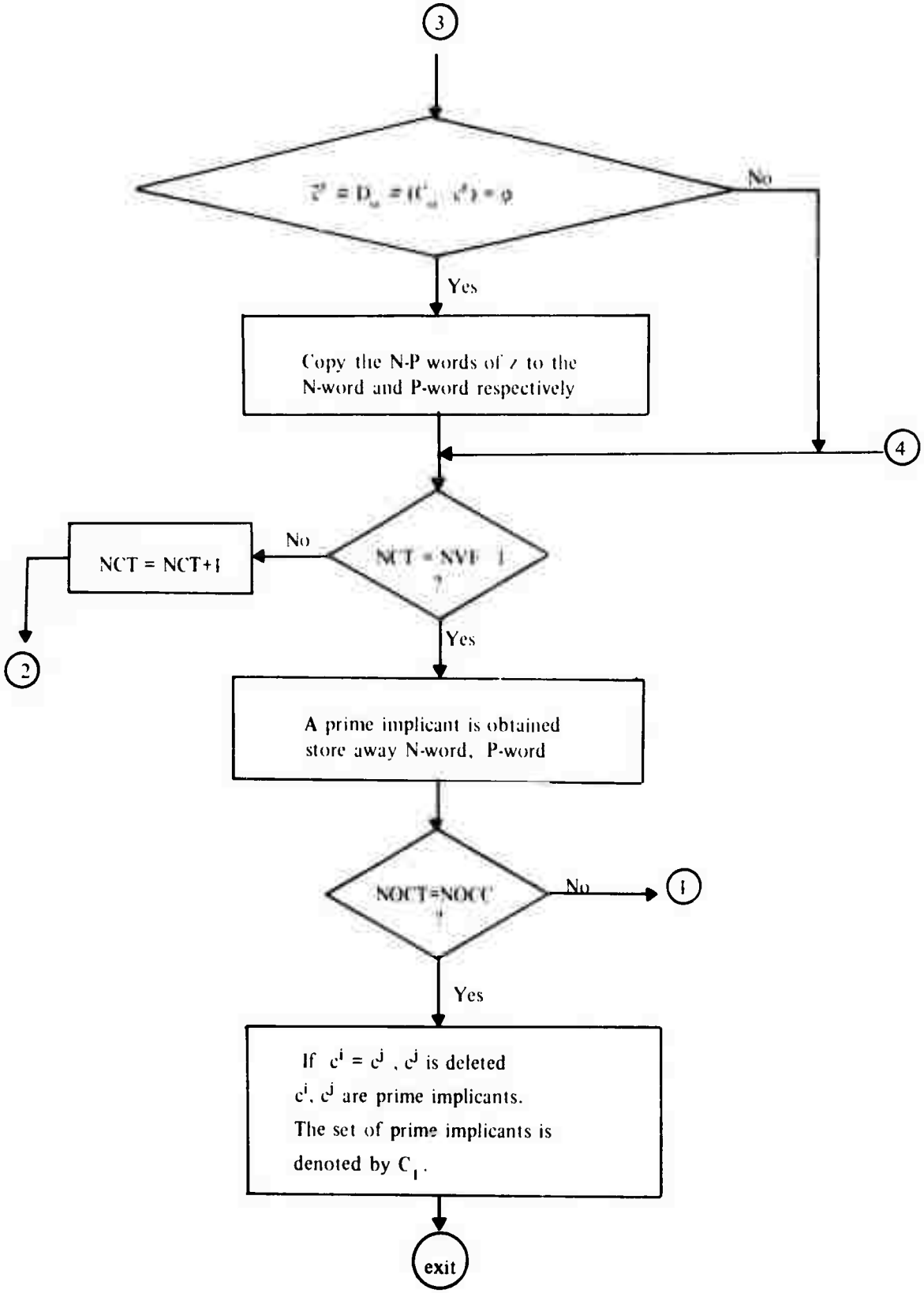
1. Cube Enlargement



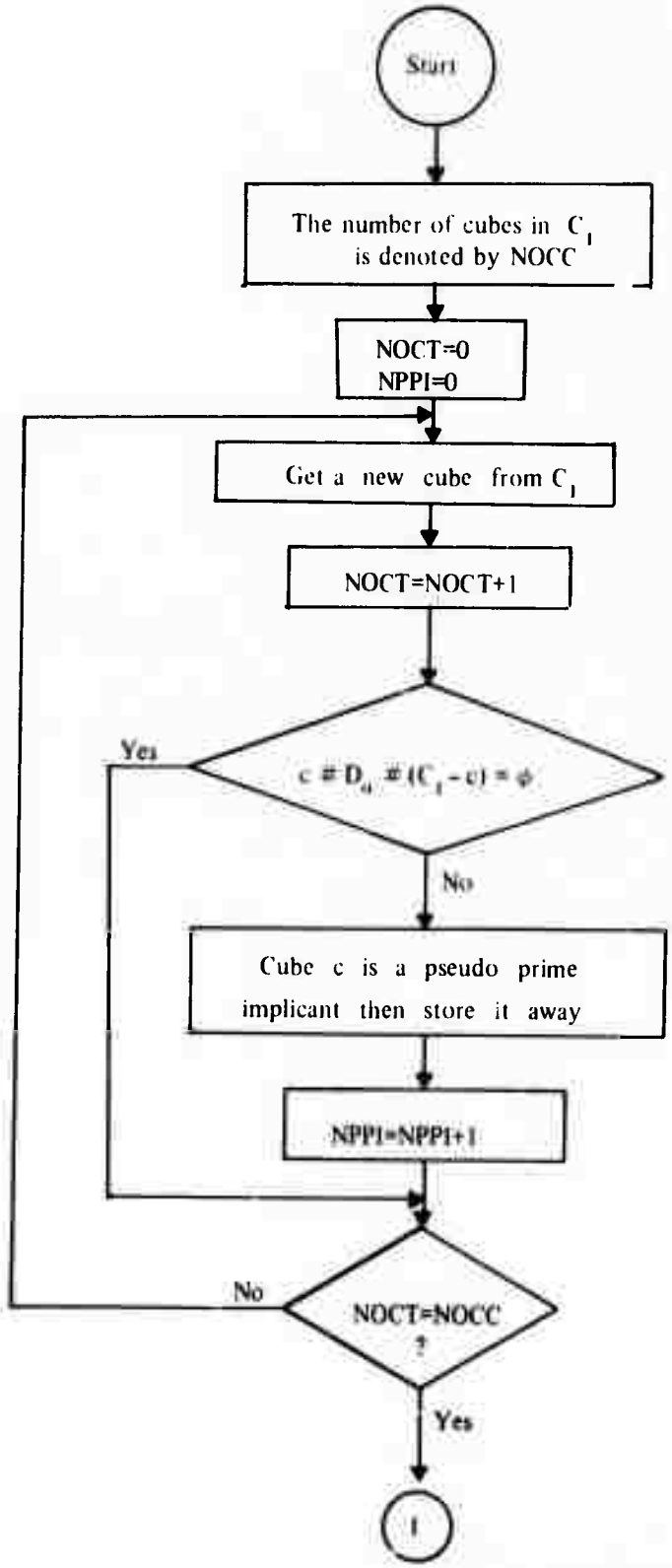


NOCT: No. of cubes tested

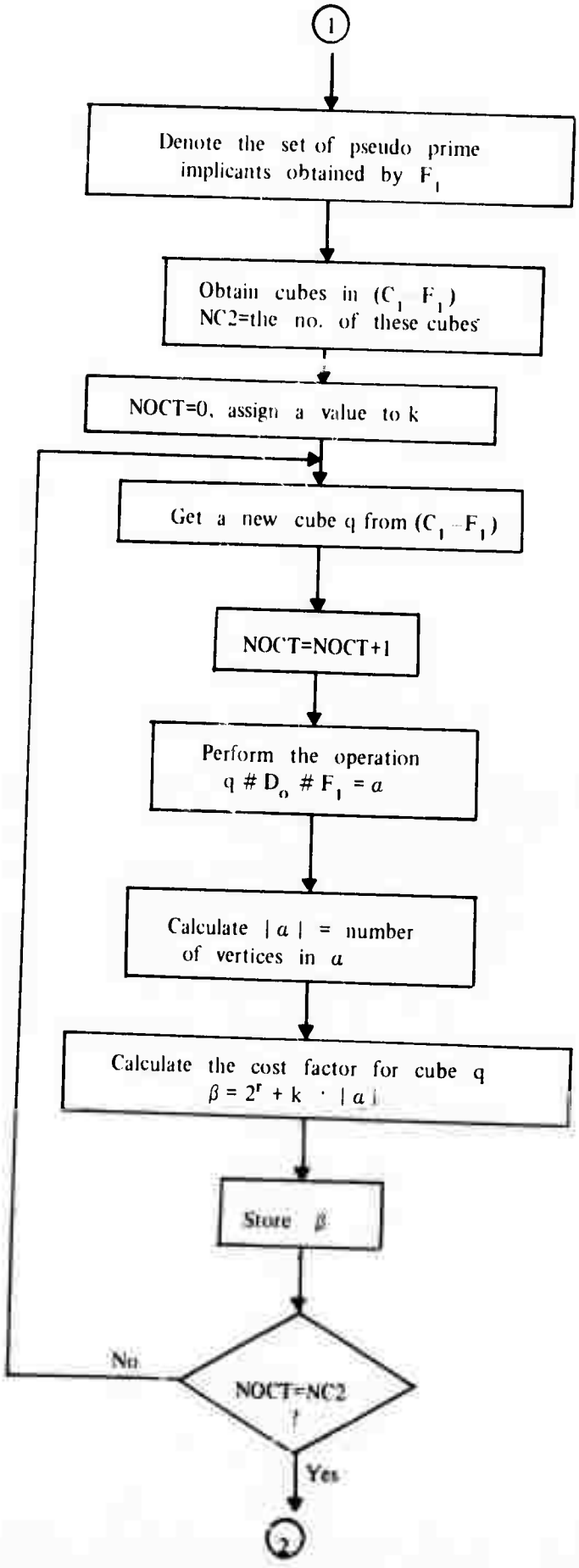
NCT: No. of components tested

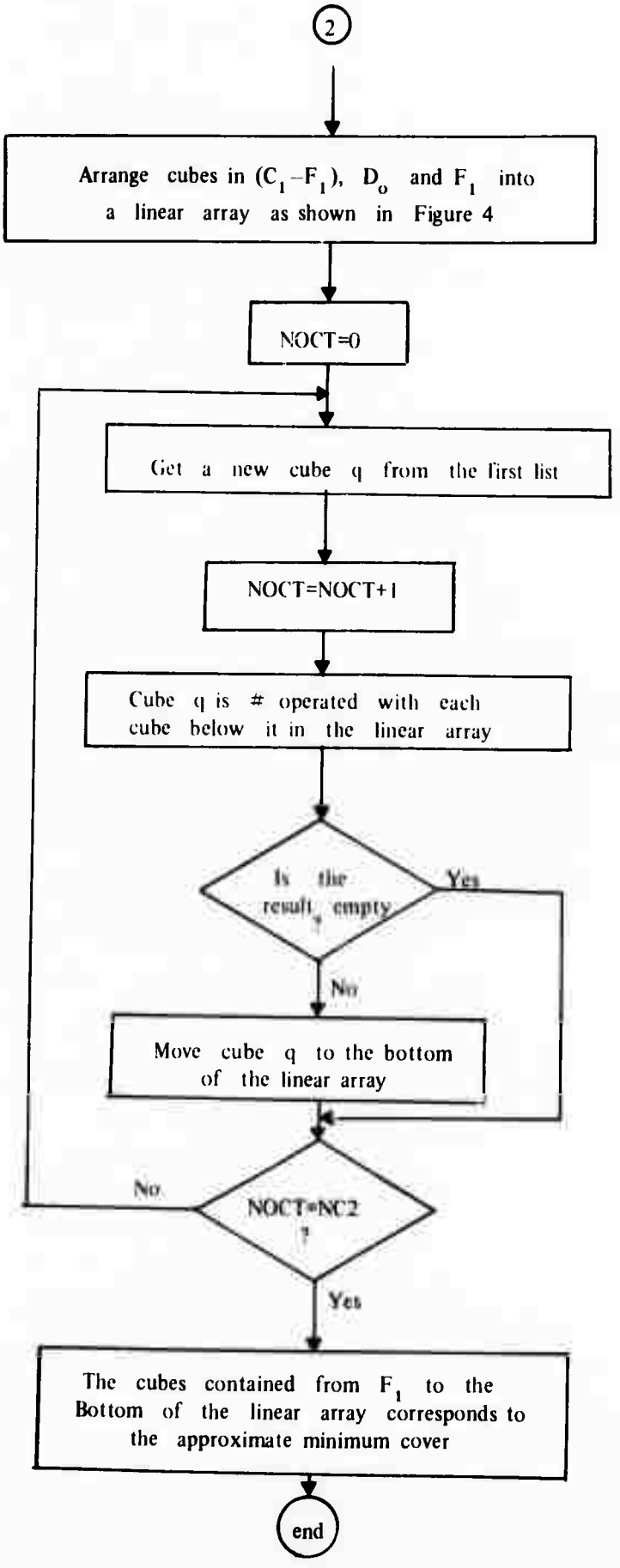

3
No
Yes
Copy the N-P words of z to the N-word and P-word respectively
4
No
NCT = NCT+1
Yes
2
A prime implicant is obtained store away N-word, P-word
No
1
Yes
If $c^i = c^j$, c^j is deleted
c^i, c^j are prime implicants.
The set of prime implicants is denoted by C_1.
exit

2. Approximate Minimum Cover



NPPI: Number of pseudo prime implicants

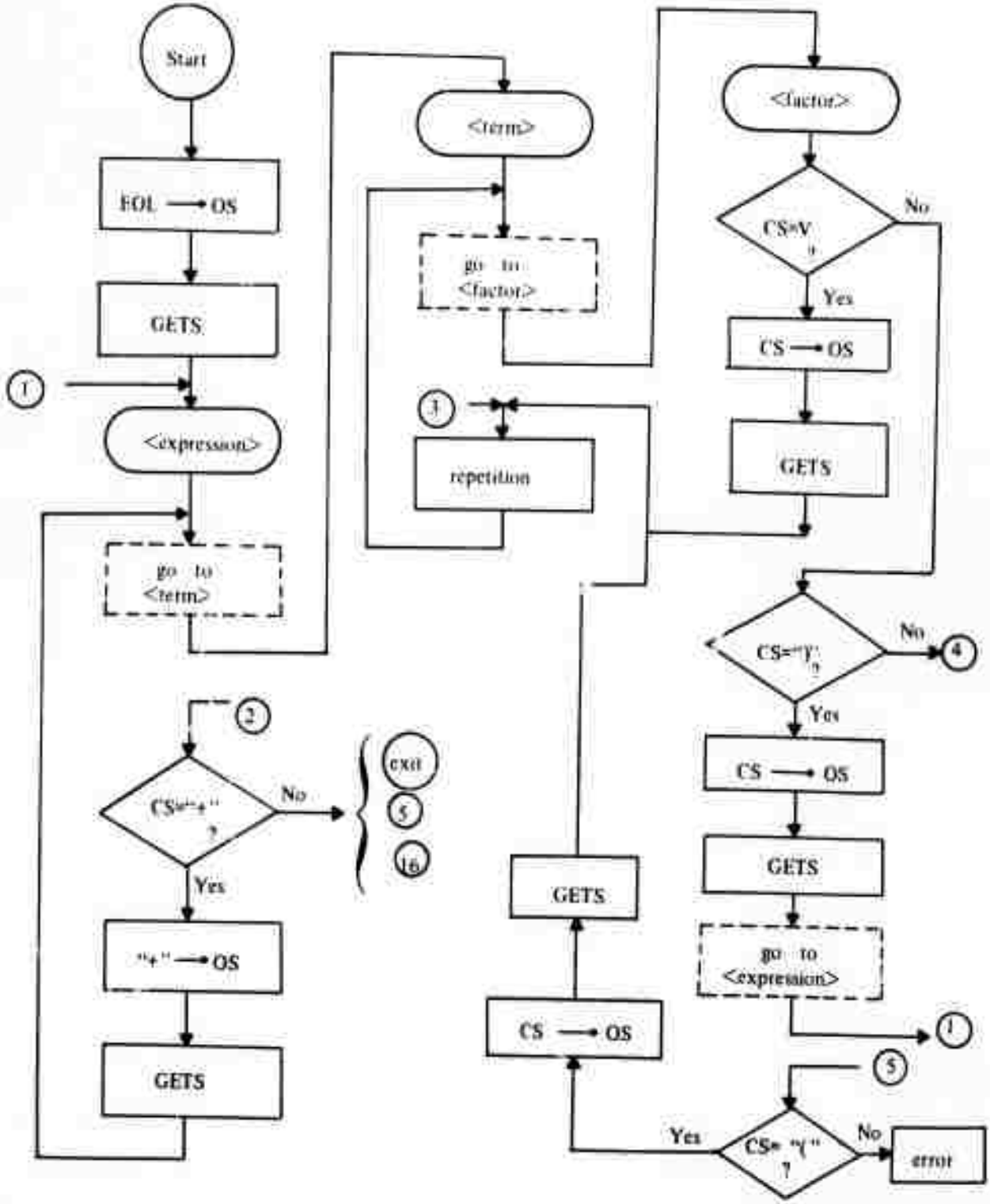

1
Denote the set of pseudo prime implicants obtained by F_1
Obtain cubes in $(C_1 - F_1)$
NC2=the no. of these cubes
NOCT=0, assign a value to k
Get a new cube q from $(C_1 - F_1)$
NOCT=NOCT+1
Perform the operation $q \# D_o \# F_1 = a$
Calculate $|a|$ = number of vertices in a
Calculate the cost factor for cube q $\beta = 2^r + k \cdot |a|$
Store β
NOCT=NC2
No
Yes
2

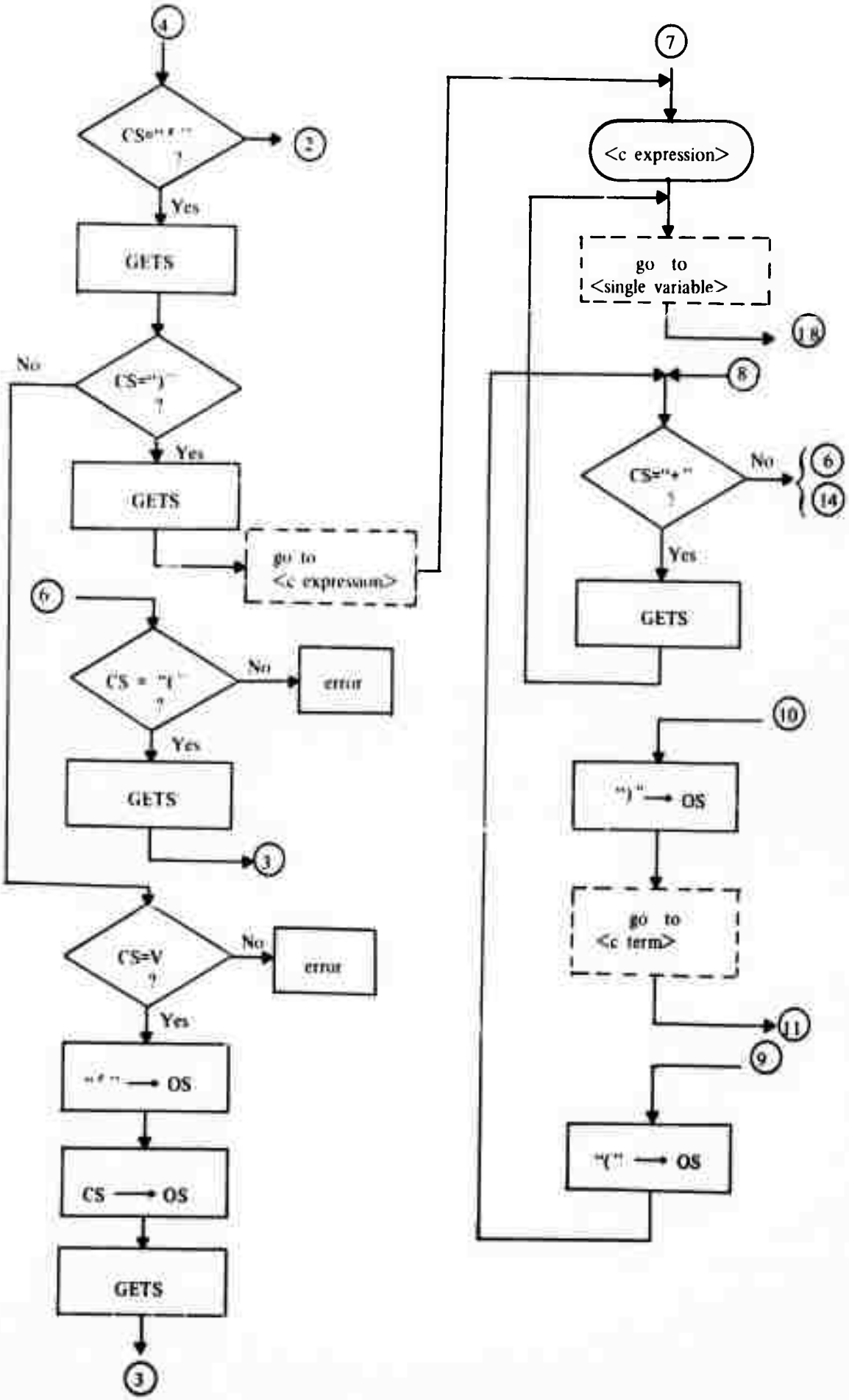

2
Arrange cubes in (C_1-F_1), D_o and F_1 into a linear array as shown in Figure 4
NOCT=0
Get a new cube q from the first list
NOCT=NOCT+1
Cube q is # operated with each cube below it in the linear array
Is the result empty ?
Yes
No
Move cube q to the bottom of the linear array
NOCT=NC2 ?
No
Yes
The cubes contained from F_1 to the Bottom of the linear array corresponds to the approximate minimum cover
end

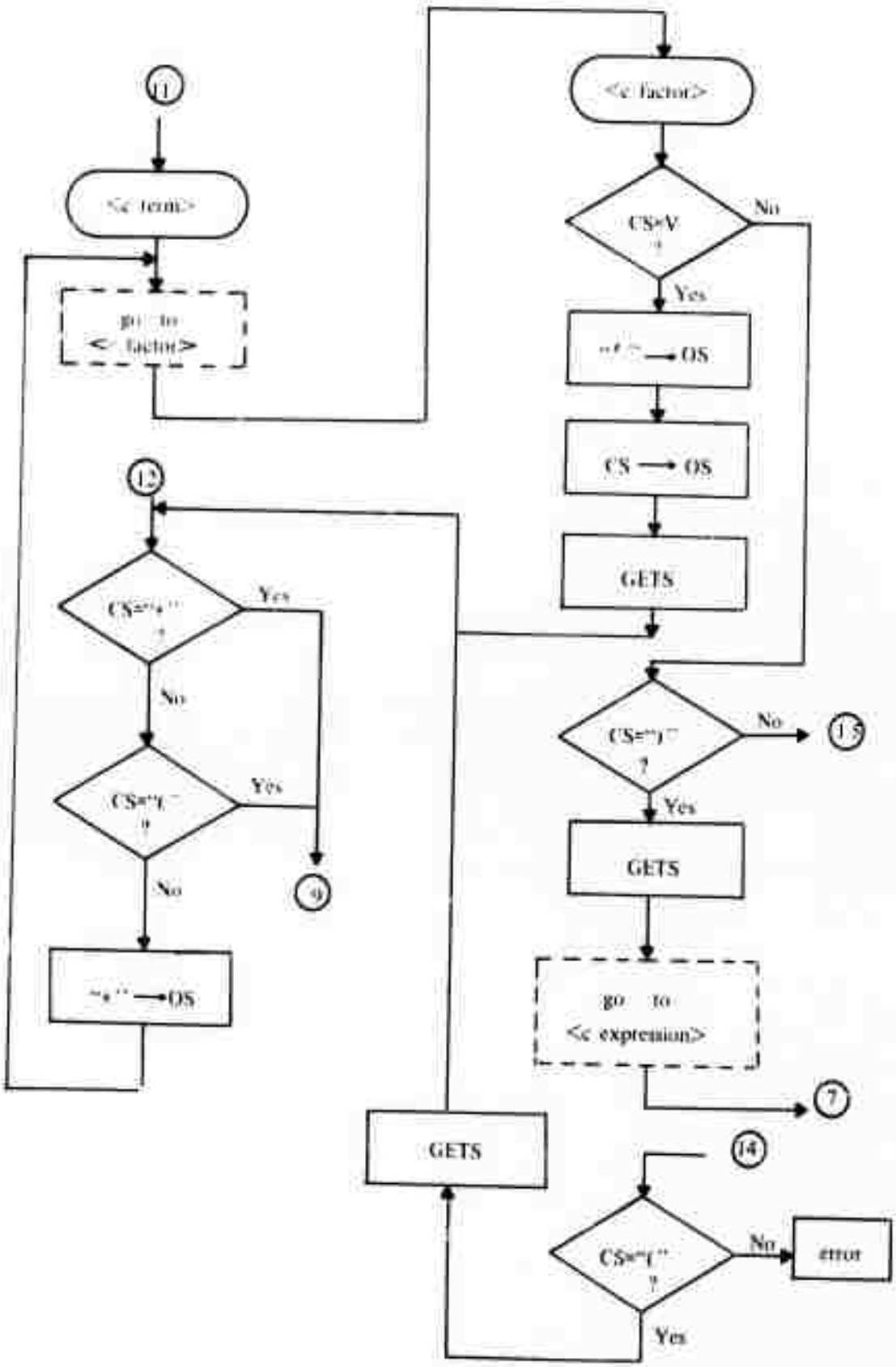
APPENDIX 7.3

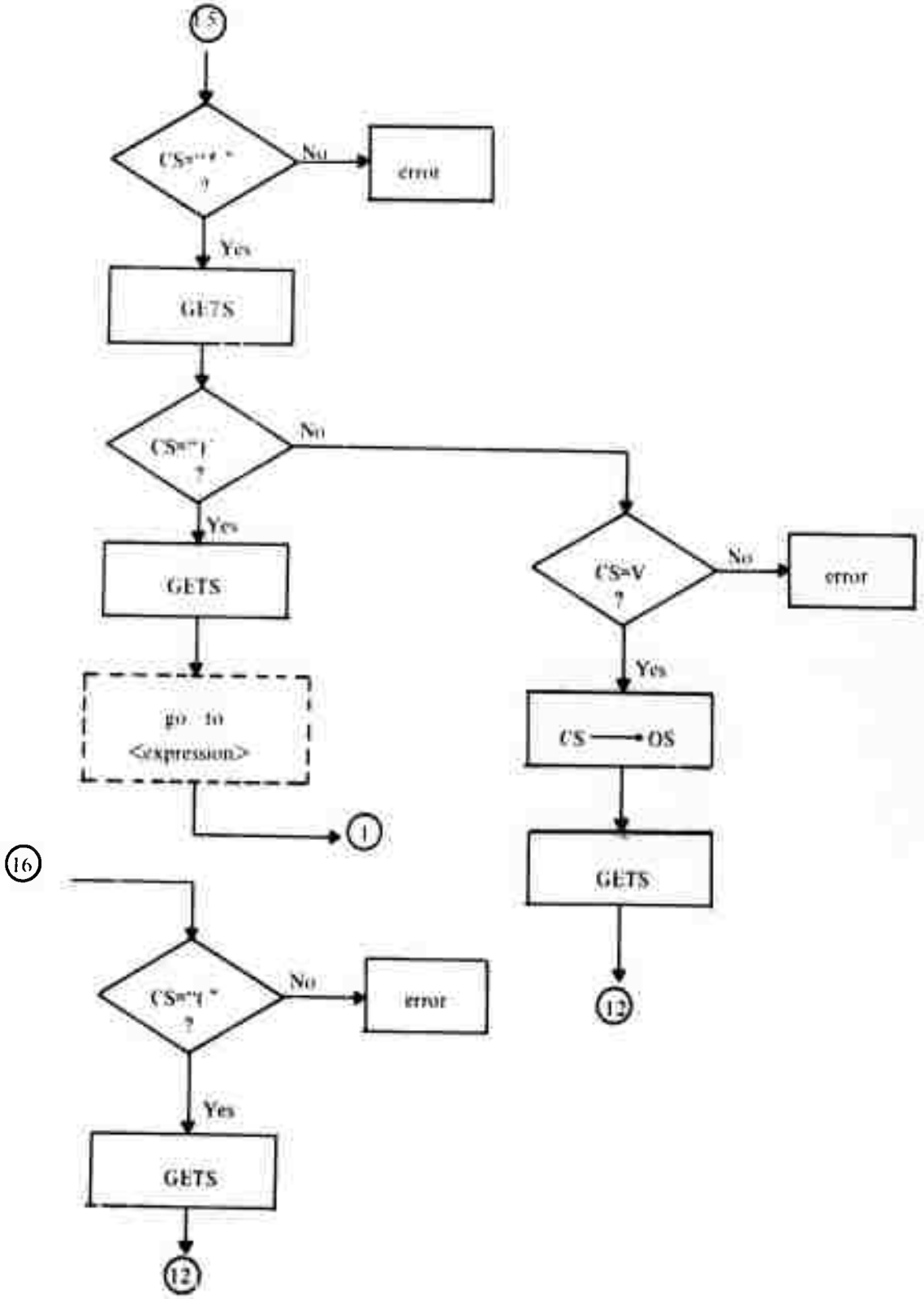
The Flow Chart for Boolean Expression Expansion

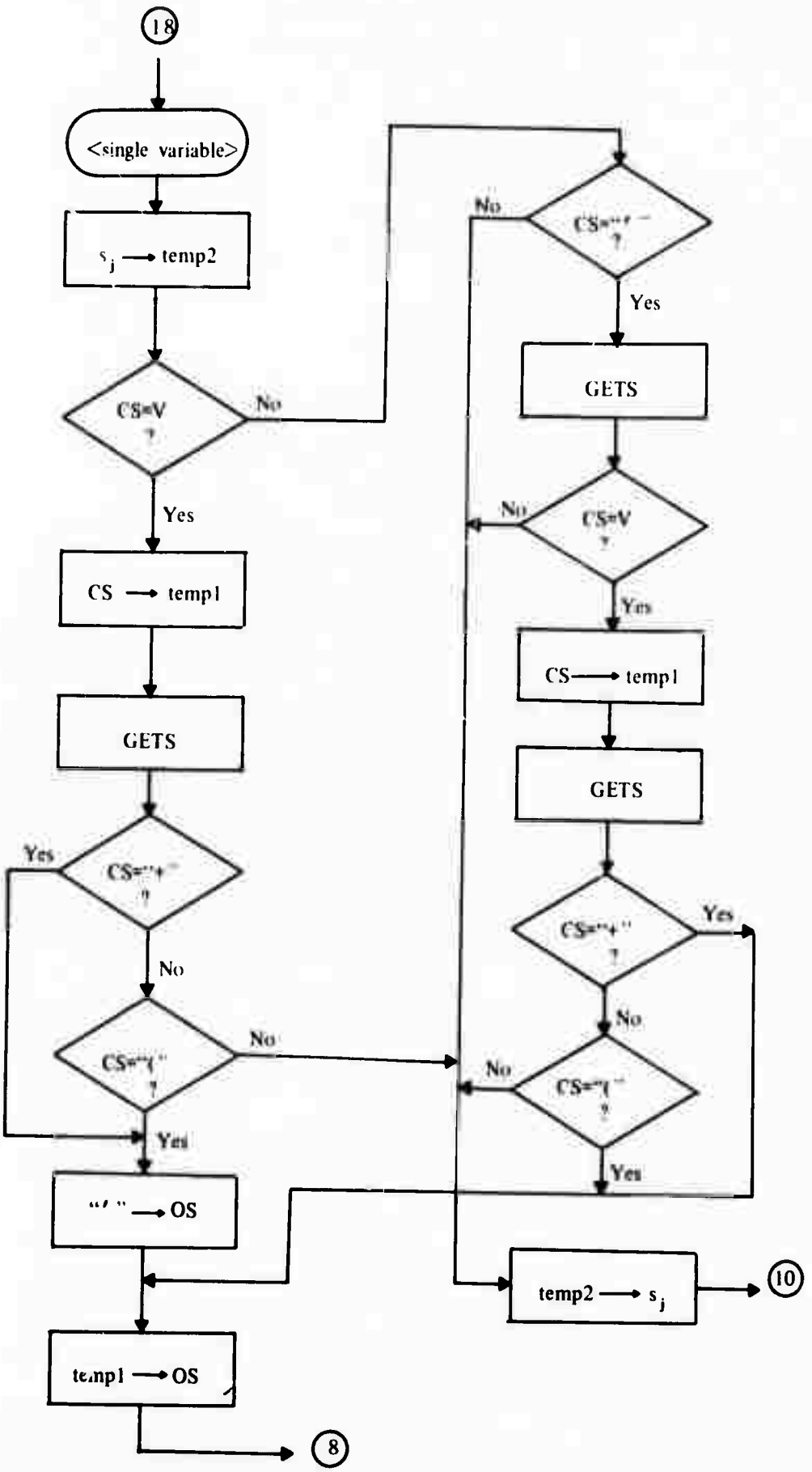
1. Applying DeMorgan's Theorems




4
CS="+" ?
2
Yes
GETS
No
CS="(" ?
Yes
GETS
go to
<c expression>
7
<c expression>
go to
<single variable>
18
8
CS="+" ?
No
6
14
Yes
GETS
6
CS = "(" ?
No
error
Yes
GETS
3
CS=V ?
No
error
Yes
"+" → OS
CS → OS
GETS
3
10
")" → OS
go to
<c term>
11
9
"(" → OS


11
<c factor>
<c term>
CS=V
?
No
Yes
go to
<c factor>
"f" → OS
CS → OS
GETS
12
CS="+"
?
Yes
No
CS="("
?
Yes
9
No
"+" → OS
CS=")"
?
No
15
Yes
GETS
go to
<c expression>
7
GETS
14
CS="("
?
No
error
Yes


15
CS="e"
?
No
error
Yes
GETS
CS="("
?
No
Yes
GETS
go to
<expression>
1
CS=V
?
No
error
Yes
CS → OS
GETS
12
16
CS=")"
?
No
error
Yes
GETS
12


18
<single variable>
s_j → temp2
CS=V ?
No
Yes
CS → temp1
GETS
CS="+" ?
Yes
No
CS="(" ?
No
Yes
" " → OS
temp1 → OS
8
CS="-" ?
No
Yes
GETS
CS=V ?
No
Yes
CS → temp1
GETS
CS="+" ?
Yes
No
CS="(" ?
No
Yes
temp2 → s_j
10

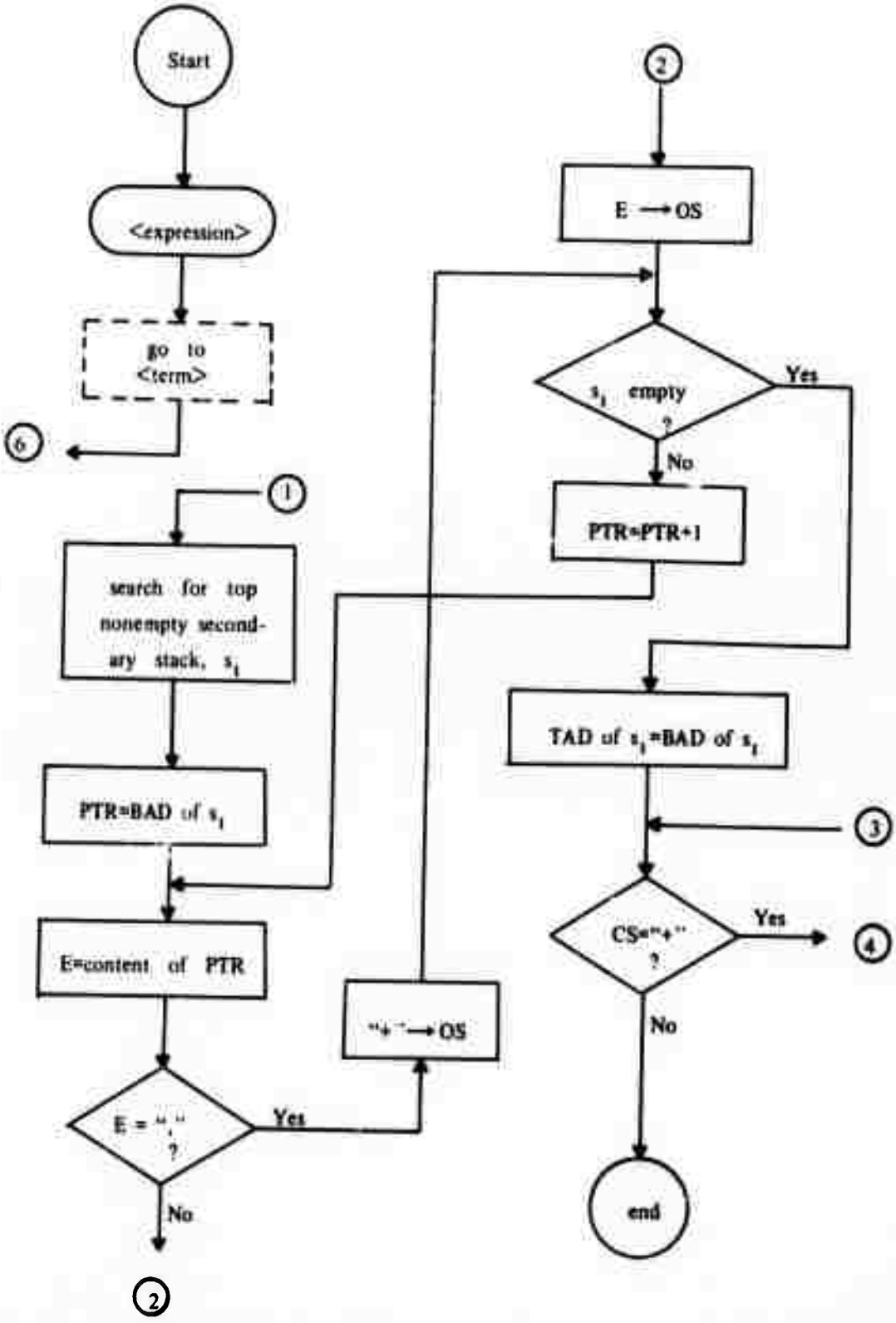
Notations

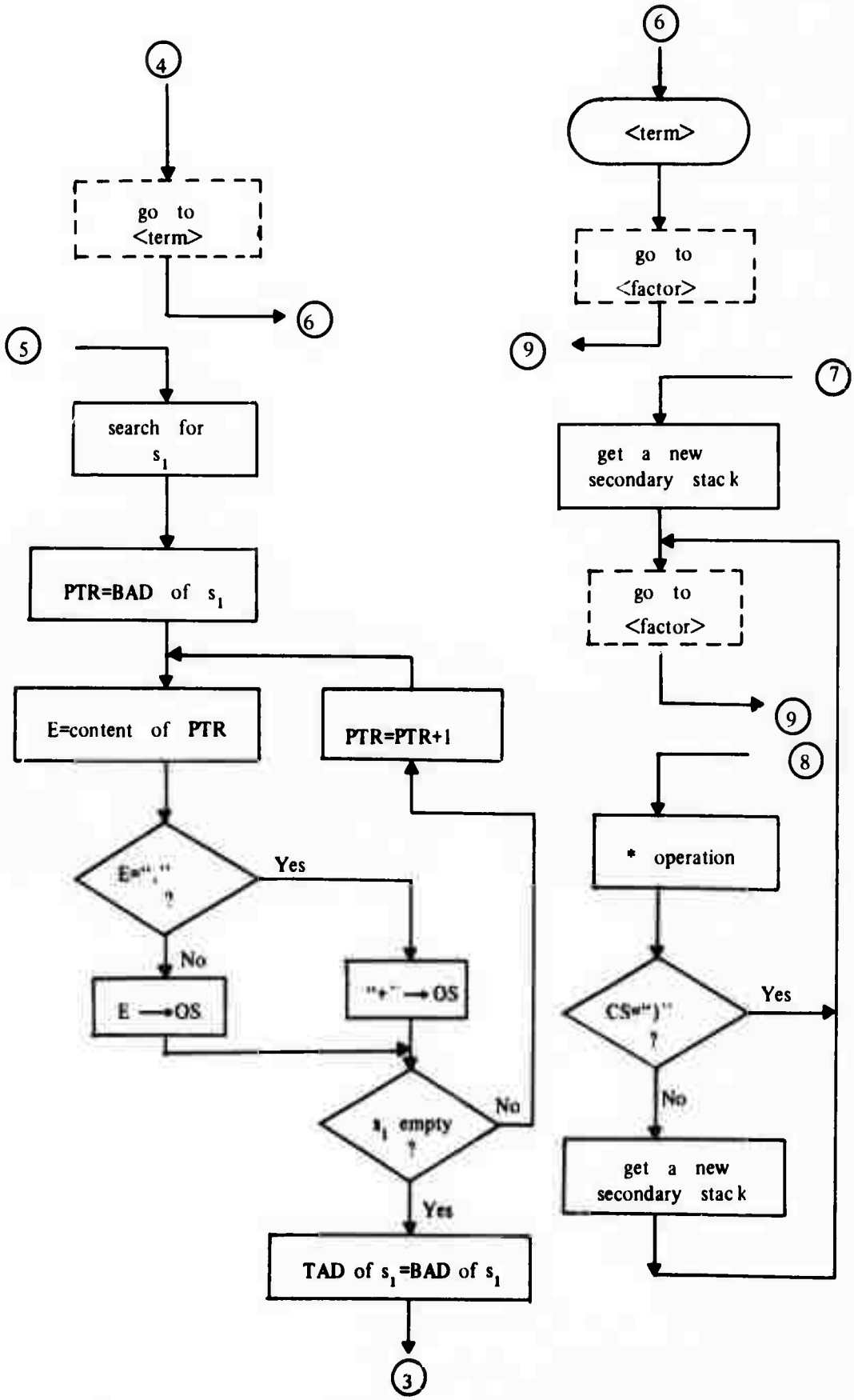
EOL:	Terminating symbol of a Boolean expression
GETS:	Read next input symbol
s_j:	Pointer points to the input symbol under examination
CS:	The current symbol
+:	"OR" operator
':	"NOT" operator
V:	A variable of the Boolean expression
temp1:	A temporary storage

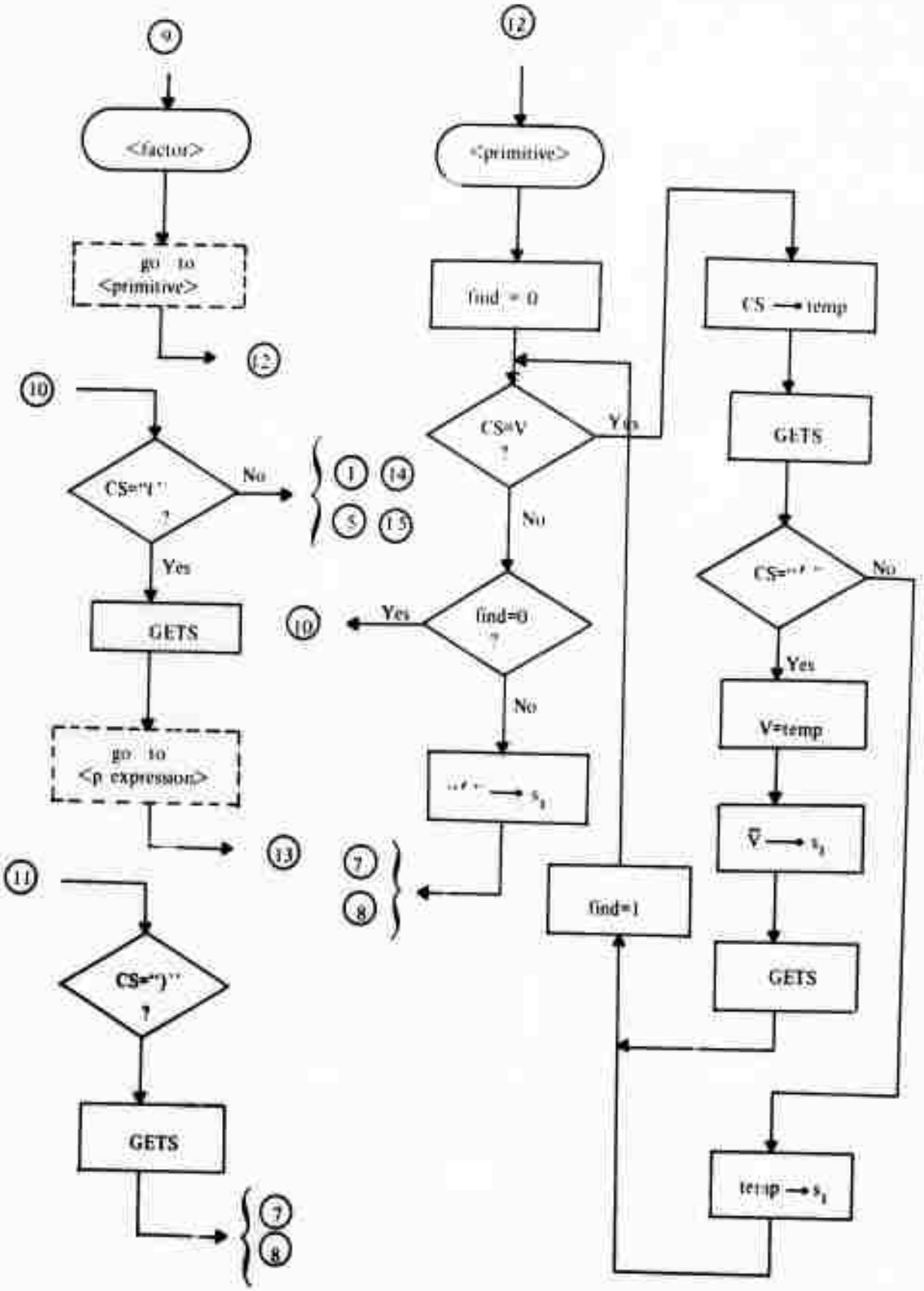
Remark

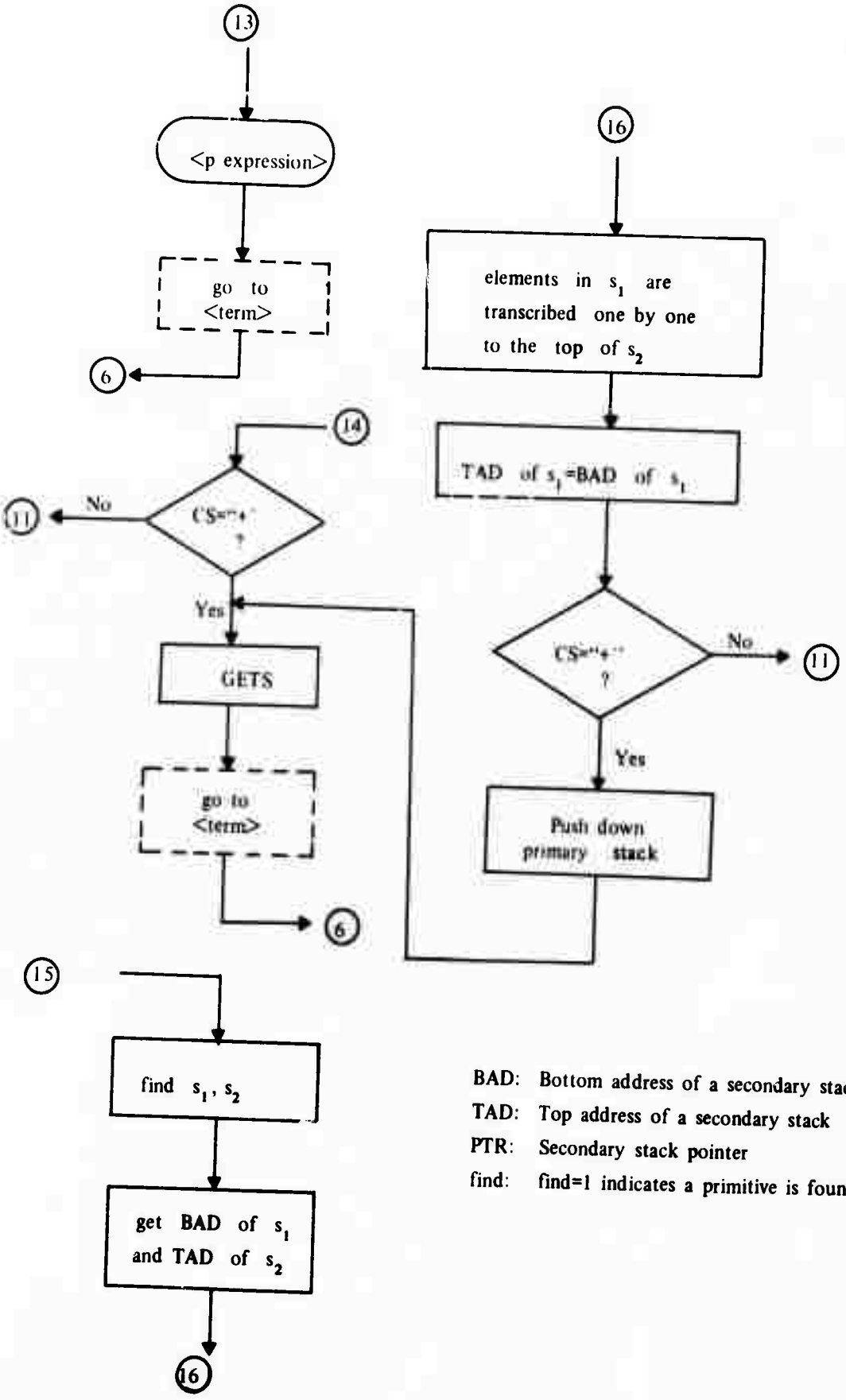
In this flow chart, an oval box indicates the entry of a syntactic class, and dot-lined rectangular box indicates the exit from one syntactic class to another syntactic class. A push-down store is set up to store return addresses. The top entry of the push-down store is always the successful return address, while the next to the top entry is always the false return address.

2. Applying Associative and Distributive Laws




4
go to <term>
6
5
search for s_1
PTR=BAD of s_1
E=content of PTR
PTR=PTR+1
E="+" ?
Yes
No
E → OS
"+" → OS
s_1 empty ?
No
Yes
TAD of s_1=BAD of s_1
3
6
<term>
go to <factor>
9
7
get a new secondary stack
go to <factor>
9
8
* operation
CS=")" ?
Yes
No
get a new secondary stack


9
<factor>
go to
<primitive>
12
10
CS="("
?
No
Yes
GETS
go to
<ρ expression>
13
11
CS=")"
?
GETS
7
8
1
14
5
15
12
<primitive>
find = 0
CS=V
?
Yes
No
find=0
?
Yes
10
No
find=1
CS → temp
GETS
No
Yes
V=temp
GETS
temp → s1



BAD: Bottom address of a secondary stack
TAD: Top address of a secondary stack
PTR: Secondary stack pointer
find: find=1 indicates a primitive is found

8. BIBLIOGRAPHY

1. Breuer, M.A., "General Survey of Design Automation of Digital Computers," *Proceedings of the IEEE,* Special Issue on Computers, (1708-1721), December 1966.

2. Dammkoehler, R.A., *A Macromodular Systems Simulator,* Technical Report No. 4, (371-376), Computer Systems Laboratory, Washington University, St. Louis, Mo., June 1967.

3. Ball, W.E., *A Macromodular Meta Machine,* Technical Report No. 4, (377-392), Computer Systems Laboratory, Washington University, St. Louis, Mo., June 1967.

4. Bashkow, T.R., and Karson, A., "A Programming System for Detection and Diagnosis of Machine Malfunctions," *IEEE Transactions on Electronic Computers,* Vol. EC-12, (10-17), February 1963.

5. Case, P.W., et al., "Solid Logic Design Automation," *IBM Journal,* Vol. 8, No. 2, (127-140), April 1964.

6. Roth, J.P., "Systematic Design of Automata," *AFIPS Conference Proceedings,* F.J.C.C. 1965, Vol. 27, Part 1, (1093-1100), Spartan Books, WAshington, D.C.

7. Schneider, P.R., and Dietmeyer, D.L., "A Subroutine Set for Automation of Logic Circuit Design," *Proceedings of the COMMON Anniversary Meetings,* New Orleans, Louisiana, November 1968.

8. Ledley, R.S., *Digital Computer and Control Engineering,* McGraw-Hill Book Company, New York, (320-427), 1960.

9. Hamblin, C.L., "Translation to and from Polish Notation," *Computer Journal,* (210-213), October 1962.

10. Cardwell, S.H., *Switching Circuits and Logical Design,* John Wiley & Sons, New York, (158-162), 1967.

11. Harada, K., *A Method Listing All Possible Permutations by Referring Hamiltonian Paths,* Technical Memorandum No. 92, Computer Systems Laboratory, Washington University, St. Louis, Mo., January 1970.

12. Quine, W.V., "The Problem of Simplifying Truth Functions," *Am. Math. Monthly,* Vol. 59, (521-531), October 1952.

13. McCluskey, E.J. Jr., *Minimization of Boolean Functions,* Bell Systems Tech. J., Vol. 35, (1417-1444), November 1956.

14. Karnaugh, M., "The Map Method for Synthesis of Combinational Logic Circuits," *Commu. and Electronics,* Trans. AIEE, Part I, Vol. 72, November 1953.

15. Roth, J.P., "Combinational Topological Methods in the Synthesis of Switching Circuits," *Proc. Intern Symp. on the Theory of Switching*, Harvard University, Cambridge, Mass., April 1957.

16. Miller, R.E., *Switching Theory, Vol. 1: Combination Circuits*, John Wiley & Sons, Chapter 3, 1965.

17. Gazale, M.J., *Irredundant Disjunction and Conjunctive Forms of a Boolean Function*, IBM J. Research and Develop., Vol. 1, (171-176), April 1957.

18. Mott, T.H., "Determination of the Irredundant Normal Forms of a Truth Function by Iterated Consensus of the Prime Implicants," *IRE Trans. on Electronic Computers.*, Vol. EC-9, (245-252), June 1960.

19. Quine, W.V., "A Way to Simplify Truth Functions," *Am. Math. Monthly*, Vol. 62, No. 9, (627-631), November 1955.

20. Naur, P., et al., "Revised Report on the Algorithmic Language ALGOL 60," *Comm. ACM*, Vol. 6, (1-17), January 1963.

21. Farber, D.J., et al., "SNOBOL: A String Manipulation Language," *J. ACM*, Vol. 11, (21-30), January 1964.

22. Tobey, R.G., Bobrow, R.J., and Zilles, S.N., "Automatic Simplification in FORMAC," *Proceedings – F.J.C.C. 1965*, (37-53).